ConvexOS Man Pages for Users

Second Edition

CONVEX COMPUTER CORPORATION

CONVEX Computer Corporation
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America
(214)497-4000

ConvexOS Man Pages for Users

Order No. DSW-331

Second Edition
December 1991

CONVEX Press
Richardson, Texas USA

ConvexOS Man Pages for Users

Order No. DSW-331

Printed in the United States of America

Revision information for

ConvexOS Man Pages for Users

Edition	Document No.	Description
Second	710-004030-003	Released with ConvexOS V10.0, December 1991.
First	710-004030-002	Initial release, April 1991.

Contents

Section 7 Text processing/terminal environments

Index

Using ConvexOS man pages

What are man pages?

The ConvexOS man pages, online and hardcopy, are a reference for all of the commands available to ConvexOS users. Each command has its own "man page," which can be one or more pages in length.

The ConvexOS man pages are organized into seven sections; commands for similar functions are grouped together. This book contains two of those sections, Section 1 and Section 7. (Traditionally, Section 6 of the man pages is allotted to games. Because CONVEX does not sell games, the ConvexOS man pages do not contain Section 6.)

How is this book organized?

This book is divided into the following sections:

- Section 1 man pages—Describe commands of general utility and commands for communication with other systems.
- Section 7 man pages—Describe miscellaneous commands, primarily in the areas of text processing and terminal environments.

Where to find other ConvexOS man pages

ConvexOS Man Pages for Programmers contains Section 2 through Section 5 of the ConvexOS man pages. These sections describe:

- Section 2 man pages—System calls and error numbers
- Section 3 man pages—Various library functions
- Section 4 man pages—Special files, related driver functions, and networking support
- Section 5 man pages—The format of various files

Section 8 man pages are found in *ConvexOS Man Pages for System Managers*. The Section 8 man pages describe various system management tools.

How to use ConvexOS man pages

The ConvexOS man pages are available in two formats: the hardcopy man pages contained in this book and the online man pages you can access by using the ConvexOS `man` command.

Using hard copy man pages

The ConvexOS man pages contained in this book are divided into Section 1 and Section 7 and ordered alphabetically within each section.

Using online man pages

`man` is a program that formats and displays information from the hard copy man pages. When invoked in the simplest way, without any options and without a topic, it displays the corresponding manual page formatted with `nroff`. Access online man pages by entering

```
% man entryname
```

where *entryname* is the name of the man page to be displayed. You can print hard copies of online man pages by entering

```
% man -t
```

For more information on `man` options, refer to the man(1) man page.

Format of man pages

The contents of each man page is divided into subsections (not all of which are relevant to each entry):

NAME	Lists exact name of command or subroutine and describes its purpose
SYNOPSIS	Summarizes use of command or subroutine being described
DESCRIPTION	Discusses use of command or subroutine
FILES	Names files built into the program
SEE ALSO/REFERENCES	Points to related information
DIAGNOSTICS/ERRORS	Discusses diagnostic messages the system produces
BUGS/NOTES	Explains known bugs or deficiencies and any known solutions

When you can't find the man page you need

One man page can document several commands. For instance, if you enter

```
% man moncontrol
```

the moncontrol(3) man page appears on your screen. However, there is no separate hard copy man page for `moncontrol`. If you look it up in the index, you are referred to the monitor(3) man page, which describes both the `moncontrol` and `monitor` commands, among others.

If you cannot find a hard copy man page for a specific command, look up that command in the index—it will refer you to the man page you need.

Technical assistance

If you have questions that are not answered in this book, contact the CONVEX Technical Assistance Center (TAC).

- Within the continental U.S., call 1(800)952-0379.
- From Canada, call 1(800)345-2384.
- All other locations, contact the local CONVEX office.

Section 1
General utilities/ communication

NAME

intro – introduction to commands

DESCRIPTION

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

(1) Commands of general utility.

(1C) Commands for communication with other systems.

(1F) FORTRAN commands.

(1G) Commands used primarily for graphics and computer-aided design.

N.B.: Commands related to system maintenance appear in section 8 manual pages.

SEE ALSO

Using the Manual Pages in the Introduction.

DIAGNOSTICS

Upon termination each command returns two bytes of status, one supplied by the system giving the cause for termination, and (in the case of 'normal' termination) one supplied by the program, see *wait* (2) and *exit*(2). The former byte is 0 for normal termination, the latter is customarily 0 for successful execution, nonzero 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

adb – debugger

SYNOPSIS

adb [**–k**] [**–w**] [**–I***dir*] [*objfil* [*corfil*]]

DESCRIPTION

adb is a general-purpose debugging program. It can be used to examine files and to provide a controlled environment for the execution of ConvexOS programs.

objfil is normally an executable program file, preferably containing a symbol table; if not, the symbolic features of *adb* cannot be used although the file can still be examined. The default name for *objfil* is "a.out". *corfil* is assumed to be a core image file produced after executing *objfil*; the default name for *corfil* is "core".

The following options are interpreted by *adb*:

–k Enable kernel mode debugging. This option should be used when *corfil* is a ConvexOS crash dump or */dev/mem*.

–w Enable write mode. Both *objfil* and *corfil* are created (if necessary), then opened for reading and writing such that they can be modified using *adb*.

–I*dir* Specify command file directory. *dir* is a directory where files to be read with the $< and $<< commands (see below) will be sought; the default is */usr/lib/adb*. Note that this specifies an alternate directory – the current directory is always searched first.

adb ignores quit signals; interrupt signals cause a return to the next *adb* command.

In general, requests to *adb* are of the form:

[*address*] [, *count*] [*command*] [;]

If *address* is present, then *dot* is set to *address*. Initially, *dot* is set to **0**. For most commands, *count* specifies how many times the command will be executed. The default *count* is **1**. *address* and *count* are expressions.

The interpretation of an address depends on the context in which it is used. When a process is being debugged, addresses are interpreted in the usual way in the address space of the process. When the operating system is being debugged, either post-mortem or by using the special file */dev/mem* to interactively examine and/or modify memory, the maps are set to map the kernel virtual addresses.

EXPRESSIONS

. The value of *dot*.

\+ The value of *dot* incremented by the current increment.

^ The value of *dot* decremented by the current increment.

" The last *address* typed.

integer A number. The prefixes **0o** and **0O** ("zero oh") force interpretation in octal radix; the prefixes **0t** and **0T** force interpretation in decimal radix; the prefixes **0x** and **0X** force interpretation in hexadecimal radix. Thus 0o20 = 0t16 = 0x10 = 16. If no prefix appears, then the default radix is used; see the $*x* command. The default radix is initially hexadecimal. The hexadecimal digits are 0123456789abcdef with the obvious values. Note that a hexadecimal number whose most significant digit would otherwise be an alphabetic character must have a **0x** (or **0X**) prefix (or a leading zero if the default radix is hexadecimal).

'cccc' The ASCII value of up to 4 characters. A backslash ("\") may be used to escape a single quote ("'").

< name The value of *name*, which is either a variable name or a register name. *adb* maintains

a number of variables named by single letters or digits. See the VARIABLES section for further details. If *name* is a register name, the value of the register is obtained from the system header in *corfil*. The register names are those displayed by the $r command.

symbol A *symbol* is a sequence of uppercase or lowercase letters, underscores or digits, not starting with a digit. The backslash character ("\") can be used to escape other characters. The value of the *symbol* is taken from the symbol table in *objfil*. An initial underscore ("_") is prepended to *symbol* if needed.

symbol In C, the "true name" of an external symbol begins with an underscore (""). It might be necessary to use the underscore to distinguish the symbol from internal or hidden variables in a program.

*_symbol_* In FORTRAN, the "true name" of an external symbol begins with an underscore and ends with an underscore. It is necessary to use the trailing underscore in order to identify the symbol properly.

(*exp*) The value of the expression *exp*.

Monadic operators.

@ *exp* The contents of the location addressed by *exp* in *objfil*.

* *exp* The contents of the location addressed by *exp* in *corfil*.

– *exp* Integer negation.

~ *exp* Bitwise complement.

exp Logical negation.

Dyadic operators are left associative and are less binding than monadic operators.

e1 + *e2* Integer addition.

e1 – *e2* Integer subtraction.

e1 * *e2* Integer multiplication.

e1 % *e2* Integer division.

e1 & *e2* Bitwise conjunction.

e1 | *e2* Bitwise disjunction.

e1 # *e2* *e1* rounded up to the next multiple of *e2*.

COMMANDS

Most commands consist of a verb followed by a modifier or list of modifiers. The following verbs are available. Note that the commands "?" and "/" may be followed by "*"; see ADDRESSES for further details.

?f Locations starting at *address* in *objfil* are displayed according to the format *f*. *dot* is incremented by the sum of the increments for each format letter (q.v.).

/f Locations starting at *address* in *corfil* are displayed according to the format *f* and *dot* is incremented as for "?".

=f The value of *address* itself is displayed in the styles indicated by the format *f*. For the **i** format, "?" is displayed for the parts of the instruction that refer to subsequent words.

A *format* consists of one or more characters that specify a style of printing. While stepping through a format, *dot* is incremented by the amount appropriate for each format letter. If no format is given, the last format is used. The format letters available are:

b[*radix*]

h[*radix*]

w[*radix*]

l[*radix*] Display a byte, halfword, word, or longword according to the radix specified by *radix*. Values for *radix* are:

x	hexadecimal.
t	signed decimal.
u	unsigned decimal.
q	signed octal.
o	unsigned octal.
f	floating point.

Hexadecimal is the default when a radix is not specified. The default may be changed by using the $*x* command. Only words and longwords may be displayed in floating-point.

f Display the 32-bit value as a floating-point number. A synonym for **wf**.

F Display double precision floating-point. A synonym for **lf**.

c Display the addressed character.

C Display the addressed character using the standard escape convention where control characters are displayed as ^**X** and the delete character is displayed as ^**?**.

s Display the addressed characters until a zero character is reached.

S Display a string using the ^**X** escape convention (see **C** above).

Y Display four bytes in date format (see *ctime*(3)).

i Display as machine instructions.

a Display the value of *dot* in symbolic form. Useful in conjunction with the **i** format character. Symbols are checked to ensure that they have an appropriate type as indicated below.

/	local or global data symbol.
?	local or global text symbol.
=	local or global absolute symbol.

P Display the addressed value in symbolic form using the same rules for symbol lookup as **a**.

t When preceded by an integer, tabs to the next appropriate tab stop. For example, **8t** moves to the next 8-space tab stop.

r Display a space.

n Display a newline.

"..." Display the enclosed string.

^ *dot* is decremented by the current increment. Nothing is displayed.

+ *dot* is incremented by **1**. Nothing is displayed.

– *dot* is decremented by **1**. Nothing is displayed.

newline Repeat the previous command with a *count* of **1**. This works only for the data formatting commands listed above.

[?/]g*format value mask*

Data starting at *dot* is masked with *mask* and compared with *value* until a match is found. If no match is found, *dot* is unchanged; otherwise, *dot* is set to the matched location. If *mask* is omitted, a mask of all 1's is used. The format specifier *format* determines the amount of data that is examined each time. Values for *format* are:

b	byte
h	halfword
w	word
l	longword

[?/]=*format value ...*

Write data into memory starting at *dot*. The format specifier *format* determines the amount of data written each time.

[?/]m*n b e f*[?/]

New values for (*b*, *e*, *f*) are recorded. *b* is the beginning address of a segment, *e* is the ending address of the segment, and *f* is the offset in the file of the segment. If less than three expressions are given, the remaining map parameters are left unchanged. *n* indicates the segment map to be modified and is required. If the list is terminated by ? or /, the file (*objfil* or *corfil*, respectively) is used for subsequent requests. (*/m?* causes / to refer to *objfil*, for example).

>*name* *dot* is assigned to the variable or register named.

$*modifier* Miscellaneous commands. The available *modifiers* are:

<*f*	Read commands from the file *f*. If this command is executed in a file, further commands in the file are ignored. If *f* is omitted, the current input stream is terminated.
<<*f*	Similar to < except it can be used in a file of commands without causing the file to be closed. There is a limit to the number of << files that can be open at once. If there are two files that need to be executed in the order so that half of file1 then file2 and then the remaining half of file1 are executed, then $<<*file2* should be inserted in file1 at the place where the commands from file2 should be executed. If the second half of file1 does not need to be executed then $<*file2* should be inserted in file1.
>*f*	Write output to the file *f*, which is created if it does not exist. Any previous contents of *f* are overwritten. If *f* is omitted, output is returned to the terminal.
?	Display the process ID, the signal(s) which caused stoppage or termination, and the general register set(s) as $*R*. If a core file is being debugged, additional information is displayed – the name of the program that caused the core file, the version of the program, and the date that the core file was created.
b	Display all breakpoints and their associated counts and commands.
c	C stack backtrace. If *address* is given, it is taken as the address of the current frame instead of the contents of the frame-pointer register. If *count* is given, only the first *count* frames are displayed.
e	Display the names and values of all external variables in the current output radix.
f [*mode*]	Display or set the floating point format. If no argument is given, the

floating point mode of *adb* and the program being debugged is displayed. If mode is one of *native*, *ieee*, or *auto*, the floating point mode of *adb* is changed accordingly. If *auto* is specified, the floating point mode of *adb* changes as needed to match that of the program being debugged. Initially, the floating point mode of *adb* is *ieee* or *native* for single mode IEEE or native mode programs, respectively. If the program is a dual mode program, *adb* begins in *auto* mode.

g Get new symbol/core file names. This prompts for new *objfil* and *corfil* names in the current process, and resets the data associated with them. A file name of "–" should be used if the file should be closed and not reopened. A *newline* will cause the file to be unchanged.

i Display the names and values of all nonzero internal variables in the current output radix.

j Display the status of all jobs (debugged processes). Information typically displayed includes the process ID, thread count, IEEE floating-point mode, and run state.

k (*Kernel debugging*) Change the current kernel memory mapping. The *address* argument is the address of the new set of SDR's (segment descriptor registers) to map. If the *count* argument is specified, it is interpreted as a thread number that is to be mapped.

l Set the limit for symbol matches to *address* (default 4095).

m Display the address maps.

n Set the maximum number of processors that may be allocated to a process. This performs the same function as the *csh*(1) *limit concurrency* command. Note that the $n command has no effect on a process currently executing – it takes effect at the next *:r* command.

o Toggle the operating mode from SEQUENTIAL to CHAINED, or from CHAINED to SEQUENTIAL in the current process. This determines the hardware execution mode of the program to be debugged. It also changes the mode of a program already being debugged, and it remains effective until the next $*o* is done. The initial operating mode is SEQUENTIAL. Note that the *:s* command will not work reliably when the operating mode is CHAINED.

q Exit from *adb*.

r[*w*] Display the general registers in hexadecimal. Registers from the current thread in the current process are displayed. If $*R* is used, the general registers from all threads in the current process are displayed. If *w* is specified, the address registers are displayed in a "wide" format. That is, they are displayed in columns aligned with the scalar registers, rather than bunched together at the beginning of the line.

t [*symbol*] Display symbol type information. This command displays the value, type, and name of all symbols in the symbol table with the name of *symbol*. The information is displayed in a manner similar to that of *nm*(1). If *symbol* begins with a number, then it is assumed to be a symbol value rather than a symbol name, and all symbols with that value are displayed. If *symbol* is not specified, information about all symbols is displayed.

x Set the default input radix to *address* and report the new value. Note that *address* is interpreted in the (old) current radix. Thus 10$x never changes the default radix. To make decimal the default radix, use 0t10$x.

X Set the default output radix to *address* and report the new value. Legal values are 0 (default old style), 0t8 (octal), 0t10 (decimal), and 0t16 (hexadecimal).

a[*register*]**?***format*[*radix*]

h[*register*]**?***format*[*radix*]

s[*register*]**?***format*[*radix*]

v[*register*][**:**[*start*][**,***count*]]**?***format*[*radix*]

Generalized display register. The methods used to examine the size and radix of **a** (address), **h** (hardware communication), **s** (scalar), and **v** (vector) registers are described below. If a register number *register* is specified, only the contents of that register are displayed. Synonyms for registers a0, a6, and a7 are **sp**, **ap**, and **fp**, respectively.

The amount of data displayed from a register depends on the format specifier *format* and radix *radix*. These are interpreted in the same manner as in the search memory command *g*. The **l** format cannot be used with **a** registers, since they are only 32 bits long.

A starting element *start* and an element count *count* may be specified when **v** registers are displayed. If neither of these options is specified, the display starts at element 0 (zero).

Other registers that can be displayed in this manner are:

pc	program counter.
psw	program status word.
vl	vector length.
vml	vector merge, lower 64 bits.
vmu	vector merge, upper 64 bits.
vs	vector stride.
dot	current address.

h*register***l**

h*register***u**

Change lock state of a hardware communication register. **l** locks *register*, **u** unlocks it.

a*register***=***format value*

h*register***=***format value*

s*register***=***format value*

v*register*[**:***start*]**=***format value*

Generalized modify register. A single register is set to the specified value. The format for **a** registers must be **w**; for **h**, **s** and **v** registers, it must be **l**. As in the generalized display register, **sp**, **ap**, and **fp** are usable. Modify **pc**, **psw**, **vl**, **vml**, **vmu**, **vs**, and *dot* in the same manner.

:*modifier* Manage a process. The available *modifiers* are:

bc Set breakpoint at *address* in the current process. The breakpoint is executed *count*–1 times before causing a stop. Each time the breakpoint is

executed, the command *c* is executed.

c*signal* The current thread in the current process is continued with signal *signal* (see *sigvec*(2)). If *address* is given, the thread is continued at this address. If no signal is specified, the signal that caused the thread to stop is sent. Breakpoint skipping is the same as for *:r*.

C*signal* As for *:c* except that all threads in the current process are continued.

d Delete breakpoint at *address* in the current process.

D Delete all breakpoints in the current process.

e Set the exit disposition for the current process. By default, the exit disposition for a debugged process is to terminate the process when quitting or restarting. Using this command sets the disposition to release on quit. That is, all breakpoints are removed and the process continues running. Note that this command is not reversible, but the *:k* command causes the process to be explicitly killed.

f Brings a process/thread pair into the debug foreground. That is, if *address* is specified, it is made the current process. If *count* is specified, it is made the current thread in the current process.

i Toggle inherit mode from OFF to ON, or from ON to OFF in the current process. When inherit mode is ON, child processes created by the debugged process inherit all debug state, thus enabling them to be debugged. The initial inherit mode is OFF.

k The current process, if any, is terminated.

m Toggle scheduling mode from FIXED to DYNAMIC, or from DYNAMIC to FIXED in the current process. When scheduling mode is set to DYNAMIC, CPU's are allocated to a process on an as-needed basis; when scheduling mode is set to FIXED, all CPU's are allocated to the process each time the process is scheduled for execution by the os. The initial scheduling mode is FIXED.

r Run *objfil* as a process. If *address* is given explicitly, the program is entered at this point; otherwise, the program is entered at its standard entry point. If *count* is given explicitly, the program stops after that many breakpoints have been reached. Arguments to the process may be supplied on the same line as the command. An argument starting with < or > causes the standard input or output to be established for the command.

s*signal* As for *:c* except that the current thread is single-stepped *count* times.

S*signal* As for *:s* except that all threads in the current process are single-stepped.

! Shell command. A shell is created to execute the rest of the line following the *!*. */bin/sh* is used to execute the command if the *SHELL* environment variable is not set.

)*command* Extended commands. The remainder of the command line is interpreted as one of several extended commands:

comment Make a (non-executed) remark. Usually used in *adb* command scripts.

help Display a file (*/usr/lib/adb/helpfile*) containing information about *adb* functionality. The file is displayed through *more*(1) if the *PAGER* environment variable is not set.

static Toggle the static symbol mode from OFF to ON, or from ON to OFF in the current process. When static symbol mode is ON, static symbols are included in commands that use symbol names. Note that static symbols do not have to be unique, and using a static symbol name may generate an incorrect or unexpected result. The initial static symbol mode is OFF.

status Display status about various *adb* modes, including the current input radix, the current output radix, inherit mode, and mode of operation.

VARIABLES

adb provides a number of variables. Named variables are set initially by *adb* but are not used subsequently. Numbered variables are reserved for communication:

0 last value displayed.

9 count on the last $< or $<< command.

On entry, the following are set from the system header in *corfil*. If *corfil* does not appear to be a normal core file, then these values are set from *objfil*:

b base address of the data segment.

d data segment size.

e entry point.

m magic number.

s stack segment size.

t text segment size.

ADDRESSES

The address in a file associated with a written address is determined by a set of mappings associated with that file. Each mapping represents a segment, and appears as three numbers – *b*, *e*, and *f*. *b* is the beginning address of the segment, *e* is the ending address of the segment, and *f* is the offset in the file of the segment. The *file address* corresponding to a written *address* in one of the segments is calculated in the following manner:

$$b \leq address < e \implies file\ address = address + f - b$$

If *address* does not fall into one of the segment ranges, it is not considered legal and an error occurs.

Two initial settings of mappings are made, one for *objfil* and one for *corfil*. Both mappings are suitable for normal files. If either file is not of the kind expected, for that file, *b* is set to **0**, *e* is set to the maximum file size and *f* is set to **0**; in this way, the whole file can be examined with no address translation.

FILES

a.out
core

SEE ALSO

cc(1), csd(1) (optional product), csh(1), pattach(2), a.out(5), core(5), fc(1f) (optional product), *CONVEX ADB Debugger User's Guide*

DIAGNOSTICS

"adb" when there is no current command or format. Comments about inaccessible files, syntax errors, abnormal termination of commands, etc. Exit status is **0**, unless the last command failed or returned nonzero status.

BUGS

Since no shell is invoked to interpret the arguments of the *:r* command, the customary wild card and variable expansions cannot occur.

NAME

ansitar – read or write ANSI multifile labeled tapes

SYNOPSIS

ansitar *subcommand [tapedev] [blocksize] [linesize] [vsn] file*
ansitar *subcommand[l] [tapedev] [blocksize] [linesize] [labelsize] [vsn] file*

DESCRIPTION

ansitar reads from or writes to a magnetic tape in ANSI-labeled tape format, which is a useful format for exchange of ASCII character files with a non-ConvexOS system. Eight-bit binary is also possible, but the results are likely to be unsatisfactory in most cases.

Subcommand is any rational concatenation of compatible characters from the following groups – only one from each group:

[crtx], [bBDF], l, U, v, V, P, [f]

c Create a new multifile tape. (Destroy all previous files on tape).

r Replace files on tape without destroying any previous files.

t List all files on tape.

x Extract files from tape. Note that if multiple entries specifying the same file name are on the tape, the last one overwrites all earlier. (Also see note on "*" wild-card character).

b Use next argument as a tape block size (default: 512 bytes).

B Use next argument as a tape block size and following argument as line size (for blocking/unblocking of fixed-length records).

D *Reading.* Not required for system level 3 tape with valid HDR2 label. For system level 1 and 2, use next argument as tape block size. *Writing.* Write HDR2/EOF2 labels and variable-length records (**D** format, system level 3). The default block size is 2048 bytes. The user can alter the block size with the *b* option. The maximum line size is 511.

F *Reading.* Not required if tape has valid HDR2 label. Use **B** for ANSI system level 1 tapes. *Writing.* Same as **B**, but write HDR2/EOF2 label for automatic unblocking. (**F** format, system level 3).

l Use next argument as nonstandard label size.

U Do case conversion on alphabetic characters in filenames: uppercase to lowercase from tape to the ConvexOS operating system, lowercase to uppercase from the ConvexOS operating system to tape. Also, convert ConvexOS user names (usually lowercase) to upper case when writing to tape. (No effect when reading from tape, as the "owner" field is not looked at when reading.)

v Use *verbose* mode – Lots of information about labels, each file added to or extracted from tape is listed along with its size, and files whose names have been truncated are printed along with the truncated name.

V Use next argument (up to six alphanumeric characters) as tape *vsn*.

f The next argument is the tape unit to use (default is */dev/rmt8*).

P Read/write files in *pip* (*FILES-11*) counted format.

The arguments for **b, B, D, F, l** are unsigned decimal strings. It is probably preferable to use only uppercase alphabetics in the argument to **V**. Arguments are evaluated in the order in which the associated *subcommand* characters appear.

Example: To write a Convex file to a tape in a format that VAX/VMS will be able to read, type the following (on the Convex):

ansitar cvDUV TAPE1 *file*

Note: The "*" wild card character may be used to get a number of related files from tape, but in that case the filename must be quoted to prevent the shell from attempting to expand the wild card. When writing from ConvexOS to tape, wild cards work normally.

SEE ALSO

ANSI X3.27-1978: Magnetic Tape Labels and File Structure for Information Interchange.
ISO 1001-1979: Magnetic Tape Labels and File Structure for Information Interchange.

BUGS

Files with names longer than 17 characters will be read and placed on the tape, but the name on the tape will be truncated to 17 characters.

NAME

apply – apply a command to a set of arguments

SYNOPSIS

apply [**–a***c*] [*–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)

AUTHOR

Rob Pike

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

ar – archive and library maintainer

SYNOPSIS

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

DESCRIPTION

ar maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the loader. It can be used, though, for any similar purpose. **N.B:** This version of *ar* uses an ASCII-format archive which is portable among the various machines running UNIX. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

key is one character from the set **drqtpmx**, optionally concatenated with one or more of **vuaibclo**. *afile* is the archive file. The *names* are constituent files in the archive file. (See the WARNINGS section for information about file names longer than 15 characters.) The meanings of the *key* characters are:

d Delete the named files from the archive file.

r Replace the named files in the archive file. If the optional character **u** is used with **r**, then only those files with '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.

FILES

/tmp/v* temporaries

SEE ALSO

lorder(1), ld(1), ranlib(1), ar(5)

WARNINGS

ar truncates file names to 15 characters.

If you use an option which requires searching the archive file, such as **d** or **r**, then file name truncation is done after the search. For example, executing the sequence of *csh* commands below will result in file *test.a* containing two copies of "abcdefghijklmnopqrstuvwxyz" named "abcdefghijklmno".

```
% rm -f test.a
% echo 'here is some random text' > abcdefghijklmnopqrstuvwxyz
% ar cr test.a abcdefghijklmnopqrstuvwxyz
% ar r test.a abcdefghijklmnopqrstuvwxyz
```

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 superuser.

NAME

as – assembler for the CONVEX supercomputer instruction set

SYNOPSIS

as [*options*] *file*

DESCRIPTION

as is an assembler for the CONVEX supercomputer instruction set. The assembler accepts an input source file and produces an object file. This object file is suitable for linking by the CONVEX loader *ld*(1). Error messages are written on the standard error stream.

For information on the instruction set and the assembler's input syntax, refer to the *CONVEX Compiler Utilities User's Guide*.

as assumes that *file* has an extension of *.s* if no extension is given.

COMMAND LINE OPTIONS

–ada This option permits special characters to be used in Ada symbols. It also recognizes assembler directives that are used only by the Ada compiler.

–cvtdc3

This option is required for assembling source code that will be executed on a **CONVEX C3800**. This option converts the instruction **cvtd.w v0,v0** to the two instructions **cvtd.l v0,v0** and **cvtl.w v0,v0**.

–fi

–fn The *-fi* option specifies that constants are translated to the IEEE floating-point format and that the mode of the object file is IEEE. Only machines with IEEE hardware can assemble files with the *-fi* option. The *-fn* option specifies that constants are translated to the CONVEX native floating-point format and the mode of the object file is native. If neither the *-fi* nor the *-fn* option is included on the command line, the site default (determined with the getsysinfo(2)) is used. The site default may be set with the sysgen(8) command. You can use the *getsysinfo -v* command to see what the default floating-point mode is on your computer.

–l This option (the letter ell) generates a source listing on the standard output stream. The listing contains the following fields:

- The source line number within the input file (this may be silently modified by any *cpp*(1) *#line* directives that appear in the source file).
- The value of the location counter (".") within the current program segment.
- The binary representation of the data or instruction.
- The current input source statement.

–n This option prevents the assembler from generating product code id information at the start of the text section. The assembler generates this information by default.

–o *outfile*

This option specifies that *outfile* is the filename to contain the generated object module. The default object filename is the source filename with an extension of *.o*.

–tm *target*

This option specifies the target instruction set for the assembler. *target* can be one of the following: c32, c34, c38, c1, or c2. The first three values are used for the C3200, C3400, or C3800 Series computers, respectively. c1 and c2 indicate that the target instruction set is for a C1 or C2 Series computer, respectively. The default instruction set is the instruction set of the computer hosting the assembler.

–w This option suppresses warning messages issued by the assembler.

REFERENCES

The program listing may be paginated for printing with *pr*(1).

The format of the object file is described in *a.out*(5).

The object file may be dumped in a human-readable form with *sod*(1).

FILES

/bin/as

SEE ALSO

cpp(1), ld(1), pr(1), sod(1), a.out(5), *CONVEX Compiler Utilities User's Guide.*

BUGS

The assembler fails to generate an error for branch instructions that use index registers as part of the destination.

The assembler fails to generate an error for instructions that use an indirection operator in conjunction with an *S* register access.

NAME

at, atrun – execute commands at a later time

SYNOPSIS

at [**–c** | **–s**] [**–m**] [**–C** *single_cmd*] *time* [*day*] [*inputfile*]

at [**–c** | **–s**] [**–C** *single_cmd*] **–f** *outputfile* [*inputfile*]

DESCRIPTION

at spools away a copy of the named *inputfile* to be used as input to *sh*(1) or *csh*(1). If the **–c** flag (for *csh*(1)) or the **–s** flag (for *sh*(1)) is specified, then that shell is used to execute the job; if no shell is specified, the current environment shell is used.

The **–C** *single_cmd* option may be used to give simply a single command for inclusion in the generated shell script. If no input file name is specified and the **–C** flag has not been given, *at* prompts for commands from standard input until a CTRL-D is typed.

If the **–m** flag is specified, mail is sent to the user after the job has been run. If errors occur during execution of the job, then a copy of the error diagnostics is sent to the user. If no errors occur, then a short message is sent informing the user that no errors occurred.

The format of the spool file is as follows: A four line header that includes the owner of the job, the name of the job, the shell used to run the job, and whether mail will be set after the job is executed. The header is followed by a *bill* command to set the activity ID to the one currently active, a *umask* command to set the modes on any files created by the job, and a *cd* command to the current directory. Then *at* copies all relevant environment variables to the spool file. Since *at* uses the .cshrc (or other startup file) to start the shell, any environment variables that are set in the .cshrc overrides the current environment variables stored in the spool file. When the script is run, it uses the user and group ID of the creator of the spool file.

The *time* is 1 to 4 digits, with an optional following "A", "P", "N" or "M" for AM, PM, noon or midnight. One- and two-digit numbers are taken to be hours, three and four digits to be hours and minutes. If no letters follow the digits, a 24-hour clock time is understood.

The optional *day* is either (1) a month name followed by a day number, or (2) a day of the week; if the word "week" follows, invocation is moved seven days further off. Names of months and days may be recognizably truncated. Examples of legitimate commands are

```
at 8am jan 24
at -c -m 1530 fr week
at -s -m 1200n week
```

at programs are executed by periodic execution of the command */usr/lib/atrun* from *cron*(8). The granularity of *at* depends upon how often *atrun* is executed.

Even if the **–m** flag is not requested, a copy of any error output is sent to the user via *mail*(1).

The **–f** *outputfile* option may be used to generate a shell script file just like one that would be run by *atrun*, placing the output in the named *outputfile* rather than in the directory of files to be run automatically. This option is useful because it creates a shell script that takes care of restoring the current environment setting, but leaves the user with complete freedom to run the script, quite possibly more than just once.

FILES

/usr/spool/at	spooling area
/usr/spool/at/yy.ddd.hhhh.*	job file
/usr/spool/at/past	directory where jobs are executed from
/usr/spool/at/lasttimedone	last time *atrun* was run
/usr/lib/atrun	executor (run by *cron*(8))

SEE ALSO

atq(1), atrm(1), calendar(1), sleep(1), cron(1)

DIAGNOSTICS

Complains about various syntax errors and times out of range.

BUGS

Due to the granularity of the execution of */usr/lib/atrun*, there may be bugs in scheduling things very close to 24 hours into the future.

If the system crashes, mail is not sent to the user informing them that the job was not completed.

Sometimes old spool files are not removed from the directory, usually due to a system crash. Such files must be removed by hand.

NAME

atq – print the queue of jobs waiting to be run

SYNOPSIS

atq [**–c**] [**–n**] [*name* ...]

DESCRIPTION

atq prints the queue of jobs that are waiting to be run at a later date. These jobs were created with the *at*(1) command. With no flags, the queue is sorted in the order that the jobs execute.

Using the **–c** flag sorts the queue by the time that the *at* command was given.

Using the **–n** flag prints only the total number of files that are currently in the queue.

If a *name*(s) is provided, only those files belonging to that user(s) are displayed.

FILES

/usr/spool/at spool area

SEE ALSO

at(1), *atrm*(1), *cron*(1)

NAME

atrm – remove jobs spooled by at

SYNOPSIS

atrm [**–f**] [**–i**] [**–**] [[*job #*] [*name*]...]

DESCRIPTION

atrm removes jobs that were created with the *at*(1) command. Using the **–** flag removes all jobs belonging to the person invoking *atrm*. If a job number(s) is specified, *atrm* attempts to remove only that job number(s).

Using the **–f** flag suppresses all information regarding the removal of the specified jobs. Using the **–i** flag causes *atrm* to ask if a job should be removed; a response of "y" causes the job to be removed.

If a *name*(s) is specified, all jobs belonging to that user(s) are removed. This form of invoking *atrm* is useful only to the superuser.

FILES

/usr/spool/at spool area

SEE ALSO

at(1), *atq*(1), *cron*(1)

NAME

awk – pattern scanning and processing language

SYNOPSIS

awk [**-F***c*] [**-f** *file*] [*file*] ...

awk [**-F***c*] [*prog*] [*file*] ...

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*.

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

An input line is made up of fields separated by white space. (This default can be changed by using FS, *vide infra*.) 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, newlines, or right braces. An empty expression-list stands for the whole line. Expressions take on string or numeric values as appropriate, and are built using the operators +, –, *, /, %, and concatenation (indicated by a blank). The C operators ++, – –, +=, –=, *=, /=, and %= are also available in expressions. Variables may be scalars, array elements (denoted x[i]), or fields. Variables are initialized to the null string. Array subscripts may be any string, not necessarily numeric; this allows for a form of associative memory. String constants are quoted "...".

The *print* statement prints its arguments on the standard output (or on a file if *>file* 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 there is no argument. There are also built-in functions *exp*, *log*, *sqrt*, and *int*. *Int* truncates its argument to an integer. *substr(s, m, n)* returns the *n*-character substring of *s* that begins at position *m*. If *n* is omitted, the substring goes to the end of *s*. *index(s, t)* returns the starting position of *t* within *s*, similar to *index()* in *string(3)*. The function *sprintf(fmt, expr, expr, ...)* formats the expressions according to the *printf*(3S) format, given by *fmt*, and returns the resulting string. *n = split(s, array, sep)* splits the string *s* into *array[1]*, ..., *array[n]*. The number of elements found is returned.

Patterns are arbitrary Boolean combinations (!, ||, &&, and parentheses) of regular expressions and relational expressions. Regular expressions must be surrounded by slashes and are as in *egrep*. 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 occurrence of the second pattern.

A relational expression is one of the following:

expression matchop regular-expression
expression relop expression

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

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

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

BEGIN { FS = "c" }

or by using the **-F***c* option.

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

INTERNAL LIMITS

awk has several internal limits that may differ from *nawk*. They are:

256	Maximum line length including end-of-line.
100	Maximum number of fields.
20	Maximum number of temp files.

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 }
```

SEE ALSO

lex(1), sed(1)

A. V. Aho, B. W. Kernighan, P. J. Weinberger, "Awk – A Pattern Scanning and Processing Language" in the *ConvexOS Tutorial Papers*

BUGS

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 " " to it.

If the input record has fields separated by tabs and a field is modified, the new input record $0 will have fields separated by spaces.

If the input record has fields separated by multiple spaces and a field is modified, the new record $0 will have fields separated by a single space.

NOTES

CONVEX *awk* allows more output files (*>file* redirection) than standard *awk*. This may cause script portability problems.

NAME

basename – strip filename affixes

SYNOPSIS

basename string [suffix]

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 ` ` in shell procedures.

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

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

SEE ALSO

sh(1)

NAME

bc – arbitrary-precision arithmetic language

SYNOPSIS

bc [**–c**] [**–l**] [file ...]

DESCRIPTION

bc is an interactive processor for a language which 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 newlines 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.

For 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.

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.

FILES

/usr/lib/lib.b mathematical library
dc(1) desk calculator proper

SEE ALSO

dc(1)
L. L. Cherry and R. Morris, *BC - An Arbitrary Precision Desk-Calculator Language* in the *ConvexOS Tutorial Papers.*

BUGS

No &&, ||, or ! operators.
for statement must have all three Es.
quit is interpreted when read, not when executed.

NAME

biff – be notified if mail arrives and who it is from

SYNOPSIS

biff [**yn**]

DESCRIPTION

biff informs the system whether you want to be notified when mail arrives during the current terminal session. The command

biff y

enables notification; the command

biff n

disables it; and the command

biff

shows the current status of biff. When mail notification is enabled, the header and first few lines of the message will be printed on your screen whenever mail arrives. A "biff y" command is often included in the file *.login* or *.profile* to be executed at each login.

biff operates asynchronously. For synchronous notification use the MAIL variable of *sh*(1) or the *mail* variable of *csh*(1).

"localhost" must be defined in */etc/hosts* in order for *biff* to work properly.

SEE ALSO

csh(1), sh(1), mail(1), comsat(8C)

DIAGNOSTICS

biff exits with a status 1 when the tty owner does not have search/execute permission; otherwise, *biff exits with a status 0.*

biff will display "biff: Where are you?" when it is unable to find the terminal that standard error is attached to.

NAME

bill – change current billing account

SYNOPSIS

bill -

bill [[*group*] *activity*]

DESCRIPTION

The *bill* command controls and reports the current billing group-activity combination. In this man page, a group-activity combination will be referred to as an account.

With no arguments, *bill* reports the current account. If the current activity ID does not match any of the activity names in */etc/activities*, the activity ID will be displayed surrounded by parenthesis; for example, "(0)".

In the one or two argument forms the *bill* command changes the current billing account, after validating access to the new account by looking in */etc/actwho*. If given two arguments, the first is taken to be the group name of the new account and the second the activity name. If given one argument, *bill* uses the argument as the activity name of the new account and does not change the group.

When a user logs on, initially his activity ID is zero and his group ID is the group ID specified in file */etc/passwd*. With the single argument -, the *bill* command changes the current account to this initial setting. This can be confusing if a user has a *bill* command in his *.login* file; after logging on, the account reported when the user executes *bill* will be the account specified by the *bill* command in his *.login* file, not the account with activity ID zero and the group ID specified in */etc/passwd*.

The *bill* command does its work by setting the group ID and activity ID of its parent process. NOTE: When a *bill* command is executed from a login shell (or *.login* file), the event is logged in */usr/adm/bill-acct*. See *bill-acct*(5) for more information.

Failed attempts at changing billing accounts will optionally be logged in file */usr/adm/bill-errs*. If and only if this file exists, the *bill* command will write an ASCII record to the file whenever a user is denied access to an account. This record contains the current time, the user's current user ID, and the requested group ID and activity ID.

Most users have a single account that they do much of their work under; by placing a *bill* command in their *.login* file, users need not remember to type a *bill* command each time they log on. The system administrator should place a *bill* command in the *.login* files of users whose work must be accounted for but have no use for or are not aware of the *bill* command.

Proper use of *bill* requires the use of the */etc/group* file. This file must be regularly maintained and must contain a complete description of the groups in use.

FILES

/usr/adm/bill-acct	log of *bill* account changes from login shell
/usr/adm/bill-errs	log of failed attempts at billing
/etc/activities	activity name-ID dictionary
/etc/actwho	group-activity access control
/etc/group	group database
/etc/passwd	user database
~/.login	file to put default *bill* command in

SEE ALSO

setaid(2), activities(5), actwho(5), bill-acct(5)
passwd(5), connecttime(8), sumscripts(8),
"Accounting Reports" chapter in the *Managing ConvexOS:Operations Guide*.

NAME

binmail – send or receive mail among users

SYNOPSIS

/bin/mail [**–i**] [person] ...
/bin/mail [**–i**] **–f** file

DESCRIPTION

Note: This is the old version 7 UNIX system mail program. The default *mail* command is described in *mail*(1), and its binary is in the directory */usr/ucb*.

mail with no argument prints a user's mail, message-by-message, in last-in, first-out order. For each message, it reads a line from the standard input to direct disposition of the message.

newline Go on to next message.

d Delete message and go on to the next.

p Print message again.

– Go back to previous message.

s [*file*] ...
Save the message in the named *files* ("mbox" default).

w [*file*] ...
Save the message, without a header, in the named *files* ("mbox" default).

m [*person*] ...
Mail the message to the named *persons* (yourself is default).

EOT (control-D)
Put unexamined mail back in the mailbox and stop.

q Same as EOT.

!*command*
Escape to the Shell to do *command*.

* Print a command summary.

An interrupt normally terminates the *mail* command; the mail file is unchanged. The optional argument **–i** tells *mail* to continue after interrupts.

When *persons* are named, *mail* takes the standard input up to an end-of-file (or a line with just ".") and adds it to each *person's* "mail" file. The message is preceded by the sender's name and a postmark. Lines that look like postmarks are prepended with ">". A *person* is usually a user name recognized by *login*(1). To denote a recipient on a remote system, prefix *person* by the system name and exclamation mark (see *uucp*(1C)).

The **–f** option causes the named file, for example, "mbox", to be printed as if it were the mail file.

When a user logs in he is informed of the presence of mail.

FILES

/etc/passwd	to identify sender and locate persons
/usr/spool/mail/*	incoming mail for user *
mbox	saved mail
/tmp/ma*	temp file
/usr/spool/mail/*.lock	lock for mail directory
dead.letter	unmailable text

SEE ALSO

mail(1), write(1), uucp(1C), uux(1C), sendmail(8)

BUGS

Race conditions sometimes result in a failure to remove a lock file.

NAME

cal – print calendar

SYNOPSIS

cal [[month] year]

DESCRIPTION

Cal prints a calendar for the specified year. If a month is also specified, a calendar just for that month is printed. If **cal** is not given any arguments it will print the calendar for the current month. *Year* can be between 1 and 9999. The *month* is a number between 1 and 12. The calendar produced is that for England and her colonies.

Try September 1752.

BUGS

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

NAME

calendar – reminder service

SYNOPSIS

calendar [–]

DESCRIPTION

calendar consults the files 'calendar' and 'calendar.<hostname>' (where <hostname> is the name of the current host) 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'. If you give the month as "*" with a date, i.e. "* 1", that day in any month will do. If you give a day of the week, *calendar* will remind you once a week regardless of any date that follows. On weekends 'tomorrow' extends through Monday.

When an argument is present, *calendar* does its job for every user who has a file 'calendar' or 'calendar.<hostname>' in his login directory and sends him any positive results by *mail*(1). Normally this is done daily in the wee hours under control of *cron*(8).

The files are first run through the "C" preprocessor, */lib/cpp*, to include any other calendar files specified with the usual "#include" syntax. Included calendars will usually be shared by all users, maintained and documented by the local administration.

FILES

calendar
calendar.<hostname> (useful in directories imported by numerous hosts)
/usr/lib/calendar to figure out today's and tomorrow's dates
/etc/passwd
/tmp/cal*
/lib/cpp, egrep, sed, mail as subprocesses

SEE ALSO

at(1), cron(1), mail(1)

BUGS

Calendar's extended idea of 'tomorrow' doesn't account for holidays.

NAME

cat – catenate and print

SYNOPSIS

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

DESCRIPTION

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

cat file

displays the file on the standard output, and

cat file1 file2 >file3

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

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

The **–n** option displays the output lines preceded by lines numbers, numbered sequentially from 1. The **–b** option works similarly to the **–n** option but omits line numbers from blank lines.

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

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

NOTES

cat is able to correctly access files larger that two gigabytes in size.

SEE ALSO

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

BUGS

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

NAME

ccrypt – CONVEX encode/decode

SYNOPSIS

ccrypt [password]

DESCRIPTION

Ccrypt 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, *ccrypt* demands a key from the terminal and turns off printing while the key is being typed in. *Ccrypt* encrypts and decrypts with the same key:

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

will print the clear. The phrase:

```
cat file | ccrypt key | ccrypt key
```

is an expensive no-op.

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 unfeasible; 'sneak paths' by which keys or cleartext can become visible must be minimized.

Ccrypt implements a simple encryption scheme devised at CONVEX Computer Corporation. Methods of attack on such schemes are easier than, say, DES.

FILES

/dev/tty for typed key

SEE ALSO

crypt(3)

NOTES

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

NAME

cd – change working directory

SYNOPSIS

cd directory

DESCRIPTION

Directory becomes the new working directory. The process 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. It is therefore recognized and executed by the shells. In *csh*(1) you may specify a list of directories in which *directory* is to be sought as a subdirectory if it is not a subdirectory of the current directory; see the description of the *cdpath* variable in *csh*(1).

SEE ALSO

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

NAME

cdecl – Compose C declarations

SYNOPSIS

cdecl

DESCRIPTION

Cdecl is a program for encoding and decoding C type-declarations. It reads standard input for statements in the language described below. A transformation is made from that language to C. The results of this transformation are written on standard output.

Cdecl's scope is intentionally small. It doesn't help you figure out storage classes or initializations.

COMMAND LANGUAGE

There are four statements in the language. The "declare" statement composes a C type-declaration from a verbose description. The "cast" statement composes a C type-cast as might appear in an expression. The "explain" statement decodes a C type-declaration or cast, producing a verbose description. The "help" statement describes the others.

The following grammar describes the language. In the grammar, words in "<>" are non-terminals, bare lower-case words are terminals that stand for themselves. Bare upper-case words are other lexical tokens: NOTHING means the empty string; NAME means a C identifier; NUMBER means a string of decimal digits; and NL means the new-line character.

```
<program>   ::= NOTHING
            | <program> <stat> NL
<stat>      ::= NOTHING
            | declare NAME as <decl>
            | cast NAME into <decl>
            | explain <huh>
            | help
<decl>      ::= array of <decl>
            | array of NUMBER <decl>
            | function returning <decl>
            | function ( <namelist> ) returning <decl>
            | pointer to <decl>
            | <type>
<huh>       ::= <type> <cdecl>
            | ( <type> <abstract> ) NAME
<cdecl>     ::= <cdecl1>
            | * <cdecl>
<cdecl1>    ::= <cdecl1> ( )
            | <cdecl1> ( <namelist> )
            | <cdecl1> [ ]
            | <cdecl1> [ NUMBER ]
            | ( <cdecl> )
            | NAME
<abstract>  ::= NOTHING
            | ( )
            | ( <abstract> ) ( )
            | * <abstract>
            | <abstract> [ ]
            | <abstract> [ NUMBER ]
            | ( <abstract> )
<type>      ::= <typename> | <modlist>
            | <modlist> <typename>
```

| struct NAME | union NAME | enum NAME
<typename> ::= int | char | double | float
<modlist> ::= <modifier> | <modlist> <modifier>
<modifier> ::= short | long | unsigned
<namelist> ::= NAME | <namelist> , NAME

EXAMPLES

To declare an array of pointers to functions like malloc(3), do

declare fptab as array of pointer to function returning pointer to char

The result of this command is

char *(*fptab[])()

When you see this declaration in someone else's code, you can make sense out of it by doing

explain char *(*fptab[])()

The proper declaration for signal(2) cannot be described in *cdecl*'s language. (It can't be described in C either!) An adequate declaration for most purposes is given by

declare signal as function returning pointer to function returning int

The function declaration that results has two sets of empty parentheses. The author of such a function might wonder where to put the parameters.

declare signal as function (args) returning pointer to function returning int

provides the solution:

int (*signal(args))()

You can use *cdecl* as you create a C program with an editor like vi(1). You simply type in the "English" version of the declaration and apply *cdecl* as a filter to the line. In vi(1), type "!!cdecl<esc>".

DIAGNOSTICS

The declare and cast statements try to point out constructions that are not supported in C. In some cases, a guess is made as to what was really intended. In these cases, the C result is a toy declaration whose semantics will work only in Algol-68. Also, certain non-portable constructs are flagged. For some C processors, these declarations will work fine.

Syntax errors cause the parser to play dead until a newline is read.

SEE ALSO

Section 8.4 of the C Reference Manual.

BUGS

The pseudo-English syntax is excessively verbose.

There is a wealth of semantic checking that isn't being done.

Cdecl thinks all the declarations you utter are going to be used as external definitions. Some declaration contexts in C allow more flexibility than this. An example of this is:

declare argv as array of array of char

where *cdecl* responds with

Unsupported in C -- Inner array of unspecified size
(maybe you mean "array of pointer")
char argv[][]

NAME

cdown - controller download utility

SYNOPSIS

cdown *executable ccu_number bus_number interrupt_level*

DESCRIPTION

cdown is a program that runs on the SPU and is used to download controller firmware. It is usually invoked at boot time by autoconf.

cdown is invoked with the following arguments:

executable	The name of the controller executable.
CCU_number	The ccu number to which the controller is attached.
bus_number	The bus number to which the controller is attached.
interrupt_level	The interrupt level of the controller.

SEE ALSO

cdump(1), creset(1)

NOTES

cdown is not supported by all controllers, using the utility to download firmware onto a controller which does not support *cdown* will result in an error. Consult the specific product documentation to determine if *cdown* is supported, as well as for the procedure in which it should be used.

NAME

cdump - controller crashdump utility

SYNOPSIS

cdump *ccu_number bus_number interrupt_level*

DESCRIPTION

cdump is a program that runs on the SPU and is used to take crashdumps of controller firmware. When a controller panic occurs, a dump should be taken before the controller is restarted.

cdump is invoked with the following arguments:

CCU_number The CCU number to which the controller is attached.

bus_number The bus number to which the controller is attached.

interrupt_level The interrupt level of the controller.

The controller image produced by *cdump* is named using the following convention:

core.<ccu><bus><interrupt_level>

SEE ALSO

cdown(1), creset(1)

NOTES

cdump is not supported by all controllers, using the utility to take a crashdump of a controller which does not support *cdump* will result in an error. Consult the specific product documentation to determine if *cdump* is supported, as well as for the procedure in which it should be used.

NAME

checknotes – check for new notesfile articles

SYNOPSIS

checknotes [**-qnvs**] [**-a**name]

DESCRIPTION

Checknotes reports on the existence of new notes. The *NFSEQ* environment variable is read to determine the notesfiles to examine for new text. If *NFSEQ* environment variable is not set the default notesfile, "general" is used. *Checknotes* then looks at each of the notesfiles to determine if there are new notes. The various flags specify the method of notification. The flags are mutually exclusive.

The **-q** option specifies a message "There are new notes" when such is the case. No text is printed if there are no new notes.

Specify **-n** to get a message "There are no new notes" if this is the case. If there are new notes, no message is printed.

With the **-v** option enabled, *checknotes* prints the name of each notesfile containing new notes.

The **-s** option is silent; no output is produced.

Regardless of the option, the program exits with 0 (TRUE) if new notes exist and with 1 (FALSE) if no new notes exist.

The **-a** option specifies a ***subsequencer.*** This allows several people sharing the same signon to maintain their own sequencer files. This switch does not allow users to forge other user's sequencer files. Since the *name* is concatenated with the user name, arbitrarily long subsequencer names aren't necessarily unique. A good rule of thumb is to ensure the first 6 characters of any given user's subsequencer names are unique.

FILES

/etc/passwd	for the users name
/etc/group	for the users group(s)
/usr/spool/notes	the default notesfile data base
/usr/spool/notes/.sequencer/*user*	Sequencing timestamps for ***user.***
/usr/spool/notes/.sequencer/*user:subsequencer*	Sub-sequencing timestamps for ***user.***

SEE ALSO

notes(1),
"The Notesfile Reference Manual" in the *ConvexOS Tutorial Papers*

BUGS

-s does note suppress warning messages about non-existent notesfiles.

NAME

checknr – check nroff/troff files

SYNOPSIS

checknr [**–s**] [**–f**] [**–a**.x1.y1.x2.y2. ... .xn.yn] [**–c**.x1.x2.x3 ... .xn] [*file* ...]

DESCRIPTION

Checknr checks a list of *nroff*(1) or *troff*(1) input files for certain kinds of errors involving mismatched opening and closing delimiters and unknown commands. If no files are specified, *checknr* checks the standard input. Delimiters checked are:

(1) Font changes using \fx ... \fP.

(2) Size changes using \sx ... \s0.

(3) Macros that come in open ... close forms, for example, the .TS and .TE macros which must always come in pairs.

Checknr knows about the *ms*(7) and *me*(7) macro packages.

Additional pairs of macros can be added to the list using the **–a** option. This must be followed by groups of six characters, each group defining a pair of macros. The six characters are a period, the first macro name, another period, and the second macro name. For example, to define a pair .BS and .ES, use **–a**.BS.ES

The **–c** option defines commands which would otherwise be complained about as undefined.

The **–f** option requests *checknr* to ignore \f font changes.

The **–s** option requests *checknr* to ignore \s size changes.

Checknr is intended to be used on documents that are prepared with *checknr* in mind, much the same as *lint*. It expects a certain document writing style for \f and \s commands, in that each \fx must be terminated with \fP and each \sx must be terminated with \s0. While it will work to directly go into the next font or explicitly specify the original font or point size, and many existing documents actually do this, such a practice will produce complaints from *checknr*. Since it is probably better to use the \fP and \s0 forms anyway, you should think of this as a contribution to your document preparation style.

SEE ALSO

nroff(1), troff(1)(optional product), neqn(1), ms(7), me(7)

DIAGNOSTICS

Complaints about unmatched delimiters.
Complaints about unrecognized commands.
Various complaints about the syntax of commands.

AUTHOR

Mark Horton

BUGS

There is no way to define a 1 character macro name using **–a**.
Does not correctly recognize certain reasonable constructs, such as conditionals.

NAME

chfn – change finger entry

SYNOPSIS

chfn [username]

DESCRIPTION

chfn is used to change information about users. This information is used by the finger program, among others. It consists of the user's "real life" name, office room number, office phone number, and home phone number. *chfn* prompts the user for each field. Included in the prompt is a default value, which is enclosed between brackets. The default value is accepted simply by typing <CR>. To enter a blank field, type the word **none**. Below is a sample run:

```
Name [Biff Studsworth II]:
Room number (Exs: 18A or 17B) []: 22C
Office Phone (Ex: 223) []: 227
Home Phone (Ex: 6610379) [5771546]: none
```

chfn allows phone numbers to be entered with or without hyphens.

It is a good idea to run finger after running *chfn* to make sure everything is the way you want it.

The optional argument *username* is used to change another person's finger information. This can only be done by the superuser.

FILES

/etc/passwd, /etc/passwd.dir, /etc/passwd.pag, /etc/ptmp

SEE ALSO

chsh(1), finger(1), passwd(1), passwd(5), mkpasswd(8)

BUGS

The encoding of the office and extension information is installation dependent.

For historical reasons, the user's name, and etc. are stored in the *passwd* file.

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.

chfn will change a local finger entry, but not a finger entry in the network Yellow Pages.

Since *chfn* rebuilds the passwd database files, a large password file may produce a long wait.

NAME

chgrp – change group

SYNOPSIS

chgrp [**-Rf**] group file ...

DESCRIPTION

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

The user invoking *chgrp* must belong to the specified group and be the owner of the file, or be the super-user.

No errors, except for usage errors, are reported when the **–f** (force) option is given.

When the **–R** option is given, *chgrp* recursively descends its directory arguments setting the specified group-ID. When symbolic links are encountered, their group is changed, but they are not traversed.

SEE ALSO

chown(8), chown(2), passwd(5), group(5)

NAME

chkpnt – checkpoint a process or process family

SYNOPSIS

chkpnt [**-CFinqrvX**] [**-j**|**-p**] [**-d** *chkpnt-directory*]
[**-f** *chkpnt-file*] [**-I** *logfile*] [**-L** *logfile*]
[**-k**|**-K** *signo*] **-P** *fd*|*pid*

DESCRIPTION

chkpnt creates a checkpoint file for a running process or process family. The checkpoint file is suitable for restarting the process hierarchy with *restart*(1). Unless *chkpnt* is invoked by the superuser, all of the target processes must belong to the current user for the checkpoint to succeed.

By default, *chkpnt* creates a checkpoint file with the name *comm.pid* in the current directory, where *comm* is the last component of the command name of the process and *pid* is the process id used to identify the process to checkpoint. If multiple processes are involved in the checkpoint (see **-j** or **-r** options), each successive *comm* is appended to the previous component followed by the *.pid* extension. For example, the checkpoint file name for a command called *myprog* started from *csh* might be *csh.myprog.123* if checkpointed in the process hierarchy starting from the *csh* process.

The list of options to *chkpnt* follows:

-C Copy all regular files in use by the checkpointed process to the checkpoint directory. The file is copied to the *chkpnt-directory* with the name *comm.pid.filename.fd#* where *comm* and *pid* are the command name and process id of the oldest process that has this file open, *filename* is the last component of a path to the open file, and *fd#* is the file descriptor number that corresponds to the file. Duplicate file descriptors obtained via *dup*(), within a process and across multiple processes of a process hierarchy cause only one copy of the file to be created, using *comm*, *pid*, and *fd#* from the oldest process.

-d *chkpnt-directory*
Create checkpoint files in *chkpnt-directory*. Default is the current directory.

-f *chkpnt-file*
Use *chkpnt-file* as the checkpoint file name. If the **-r** or **-j** option is used in combination with this option, a checkpoint file for each process in the hierarchy is created by replacing the top-level command name in the hierarchy with *chkpnt-file* and appending *.comm2.pid* as required. If *chkpnt-file* contains a "/", the directory component of the path is used as the *chkpnt-directory*. If this implicit *chkpnt-directory* is different from a directory given with the **-d** option, an error is reported.

-F Force a checkpoint of the process even if non-checkpointable conditions exist. This may create a checkpoint file that will not restart correctly. *chkpnt* will print warning messages describing each non-checkpointable condition.

-i Invoke *chkpnt* in interactive mode. *chkpnt* will prompt for an action on each file descriptor providing the file descriptor number, associated pathname, and file size. Actions include copying the contents of the associated file to the checkpoint directory, checkpointing the associated file by pathname only, or ignoring the file descriptor.

-I *logfile*
Use *logfile* previously created with the **-L** option as a description of the actions to be taken on each file descriptor. File descriptors in use by the target processes that have no action in *logfile* are prompted for interactively.

-j Checkpoint the entire job hierarchy rooted at *pid*. Same as **-r**.

-k *signo*
Send the root process of a process hierarchy a signal after the checkpoint has completed successfully. Signals are specified by giving either the signal name (e.g., KILL or SIGKILL) or signal number (e.g., 9). No signal is sent if neither **-k** nor **-K** is specified. No signal is sent if the checkpoint fails for any reason.

-K *signo*
Like **-k** but send the signal to all the processes in the hierarchy.

-L *logfile*
Invoke *chkpnt* in interactive mode and log the information associated with each file descriptor in *logfile*, suitable for use as input using the **-I** option.

-n Perform all checkpointability tests on the target processes but do not actually perform the checkpoint. This is useful to determine if a process meets the criteria for a checkpointable process or to create an interactive *logfile*. If the process is checkpointable no error messages will be printed and *chkpnt* will exit with a zero status. If the process does not meet the criteria for a checkpointable process a warning will be printed for the *first* non-checkpointable condition found. If the **-F** option is used together with this flag a warning is printed for *all* non-checkpointable conditions.

-p Checkpoint only the process specified by *pid* and no other descendents. This is the default but is supplied for compatibility with other implementations.

-P *fd* Use the inherited file descriptor *fd* as a process file descriptor obtained through *pattach*(2) associated with the process to be checkpointed. The process must have been attached for exclusive use and be stopped. This option is used to support debuggers.

-q Quiet mode. Suppress warning messages that may be generated during a checkpoint. Fatal error messages are still generated.

-r Recursive checkpoint. Checkpoint the entire process hierarchy rooted at *pid*.

-v Provide verbose informational output on checkpoint.

-X Print debugging output.

NOTES

Processes using the following facilities cannot be checkpointed:

Processes that have a socket connection.

Processes holding file or record locks.

Processes running setuid to the superuser or another user.

Processes running setgid to a group outside the user's group access list.

Processes that are debugging another process or contain a process file descriptor.

Processes with more than CHKPNT_MAXREGIONS memory regions. The value for CHKPNT_MAXREGIONS is defined in <chkpnt.h>.

Processes with more than CHKPNT_MAXOPENFDS open file descriptors. The value for CHKPNT_MAXOPENFDS is defined in <chkpnt.h>.

Processes using devices that do not have checkpoint capability. Currently, devices that do support checkpointing include: terminal devices including */dev/tty*, slave side of pseudo-terminals, raw tape devices, and the null device */dev/null*.

Processes with open pipes can be checkpointed and restarted correctly if the pipe connections do not go outside of the checkpoint process hierarchy.

DIAGNOSTICS

chkpnt exits with 0 upon successful completion of the checkpoint, even if non-checkpointable conditions were ignored with the **-F** option. If a checkpoint fails, *chkpnt* exits with the value of *errno* to indicate the error.

NOTES

chkpnt is capable of checkpointing processes that have files greater than two gigabytes in size open.

SEE ALSO

restart(1), pattach(2), chkpnt(3), restart(3), chkpnt(5)
Convex Checkpoint Restart Guide

NAME

chmagic – change magic number of executable file

SYNOPSIS

chmagic –d file ...
chmagic –p file ...
chmagic –l file ...

DESCRIPTION

Chmagic changes the magic number of the specified files. This utility will only work on old-style (pre-SOFF) executables. To perform the same types of functions on SOFF files, use the loader/link editor, *ld*. The **–d** option changes the file's magic number to ZMAGIC (demand paged executable). Demand paged is the default mode of an executable file created by the link editor, *ld*(1). The **–p** option changes the file's magic number to PMAGIC (pre-paged executable). The **–l** option changes the file's magic number to LMAGIC (pre-paged, non-swapped executable). The magic numbers are defined in */usr/include/a.out.h*.

Only the superuser can execute a file whose magic number is LMAGIC.

SEE ALSO

ld(1), a.out(5)

NAME

chmod – change mode

SYNOPSIS

chmod mode file ...

DESCRIPTION

The mode of each named file is 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 effective user ID on execution
2000	set effective group ID on execution
1000	sticky bit, see *sticky*(8)
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 all, or **ugo.** If *who* is omitted, the default is *a* but the setting of the file creation mask (see umask(2)) is taken into account.

Op can be + to add *permission* to the file's mode, – to take away *permission* and = 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** (sticky – see *sticky*(8)) . Letters **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.

EXAMPLES

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

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

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.**

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

SEE ALSO

ls(1), chmod(2), stat(2), umask(2), chown(8)

NAME

chsh – change default login shell

SYNOPSIS

chsh username [shell]

DESCRIPTION

chsh changes the login shell field of the user's password file entry. The argument *shell* may be either a full path name, or the last component thereof. If no *shell* is specified, then the shell reverts to the default login shell */bin/sh*. Otherwise only shells listed in */etc/shells* can be specified unless you are the superuser. If */etc/shells* does not exist, */bin/sh* and */bin/csh* are the only shells that may be changed to.

An example use of this command would be:

chsh bill /bin/csh

FILES

/etc/passwd, /etc/passwd.dir, /etc/passwd.pag, /etc/ptmp, /etc/shells

SEE ALSO

chfn(1), csh(1), sh(1), passwd(1), passwd(5), mkpasswd(8)

BUGS

chsh will change a local shell, but not a shell in the network Yellow Pages.

Since *chsh* rebuilds the passwd database files, a large password file may produce a long wait.

NAME

ci – check in RCS revisions

SYNOPSIS

ci [options] file ...

DESCRIPTION

Ci stores new revisions into RCS files. Each file name ending in ',v' is taken to be an RCS file, all others are assumed to be working files containing new revisions. *Ci* deposits the contents of each working file into the corresponding RCS file.

Pairs of RCS files and working files may be specified in 3 ways (see also the example section of *co* (1)).

1) Both the RCS file and the working file are given. The RCS file name is of the form *path1/workfile*,v and the working file name is of the form *path2/workfile*, where *path1/* and *path2/* are (possibly different or empty) paths and *workfile* is a file name.

2) Only the RCS file is given. Then the working file is assumed to be in the current directory and its name is derived from the name of the RCS file by removing *path1/* and the suffix ',v'.

3) Only the working file is given. Then the name of the RCS file is derived from the name of the working file by removing *path2/* and appending the suffix ',v'.

If the RCS file is omitted or specified without a path, then *ci* looks for the RCS file first in the directory ./RCS and then in the current directory.

For *ci* to work, the caller's login must be on the access list, except if the access list is empty or the caller is the superuser or the owner of the file. To append a new revision to an existing branch, the tip revision on that branch must be locked by the caller. Otherwise, only a new branch can be created. This restriction is not enforced for the owner of the file, unless locking is set to *strict* (see *rcs* (1)). A lock held by someone else may be broken with the *rcs* command.

Normally, *ci* checks whether the revision to be deposited is different from the preceding one. If it is not different, *ci* either aborts the deposit (if **-q** is given) or asks whether to abort (if **-q** is omitted). A deposit can be forced with the **-f** option.

For each revision deposited, *ci* prompts for a log message. The log message should summarize the change and must be terminated with a line containing a single '.' or a control-D. If several files are checked in, *ci* asks whether to reuse the previous log message. If the std. input is not a terminal, *ci* suppresses the prompt and uses the same log message for all files. See also **-m**.

The number of the deposited revision can be given by any of the options **-r**, **-f**, **-k**, **-l**, **-u**, or **-q** (see **-r**).

If the RCS file does not exist, *ci* creates it and deposits the contents of the working file as the initial revision (default number: 1.1). The access list is initialized to empty. Instead of the log message, *ci* requests descriptive text (see **-t** below).

–r[*rev*] assigns the revision number *rev* to the checked-in revision, releases the corresponding lock, and deletes the working file. This is also the default.

If *rev* is omitted, *ci* derives the new revision number from the caller's last lock. If the caller has locked the tip revision of a branch, the new revision is appended to that branch. The new revision number is obtained by incrementing the tip revision number. If the caller locked a non-tip revision, a new branch is started at that revision by incrementing the highest branch number at that revision. The default initial branch and level numbers are **1**. If the caller holds no lock, but he is the owner of the file and locking is not set to *strict*, then the revision is appended to the trunk.

If *rev* indicates a revision number, it must be higher than the latest one on the branch to which *rev* belongs, or must start a new branch.

If *rev* indicates a branch instead of a revision, the new revision is appended to that branch. The level number is obtained by incrementing the tip revision number of that branch. If *rev* indicates a non-existing branch, that branch is created with the initial revision numbered *rev.1*.

Exception: On the trunk, revisions can be appended to the end, but not inserted.

-f[*rev*] forces a deposit; the new revision is deposited even it is not different from the preceding one.

-k[*rev*] searches the working file for keyword values to determine its revision number, creation date, author, and state (see *co* (1)), and assigns these values to the deposited revision, rather than computing them locally. This option is useful for software distribution. A revision that is sent to several sites should be checked in with the **-k** option at these sites to preserve its original number, date, author, and state.

-l[*rev*] works like **-r**, except it performs an additional *co -l* for the deposited revision. Thus, the deposited revision is immediately checked out again and locked. This is useful for saving a revision although one wants to continue editing it after the checkin.

-u[*rev*] works like **-l**, except that the deposited revision is not locked. This is useful if one wants to process (e.g., compile) the revision immediately after checkin.

-q[*rev*] quiet mode; diagnostic output is not printed. A revision that is not different from the preceding one is not deposited, unless **-f** is given.

-m*msg* uses the string *msg* as the log message for all revisions checked in.

-n*name* assigns the symbolic name *name* to the number of the checked-in revision. *Ci* prints an error message if *name* is already assigned to another number.

-N*name* same as **-n**, except that it overrides a previous assignment of *name*.

-s*state* sets the state of the checked-in revision to the identifier *state*. The default is *Exp*.

-t[*txtfile*] writes descriptive text into the RCS file (deletes the existing text). If *txtfile* is omitted, *ci* prompts the user for text supplied from the std. input, terminated with a line containing a single '.' or control-D. Otherwise, the descriptive text is copied from the file *txtfile*. During initialization, descriptive text is requested even if **-t** is not given. The prompt is suppressed if std. input is not a terminal.

DIAGNOSTICS

For each revision, *ci* prints the RCS file, the working file, and the number of both the deposited and the preceding revision. The exit status always refers to the last file checked in, and is 0 if the operation was successful, 1 otherwise.

FILE MODES

An RCS file created by *ci* inherits the read and execute permissions from the working file. If the RCS file exists already, *ci* preserves its read and execute permissions. *Ci* always turns off all write permissions of RCS files.

FILES

The caller of the command must have read/write permission for the directories containing the RCS file and the working file, and read permission for the RCS file itself. A number of temporary files are created. A semaphore file is created in the directory containing the RCS file. *Ci* always creates a new RCS file and unlinks the old one. This strategy makes links to RCS files useless.

IDENTIFICATION

Author: Walter F. Tichy
Revision Number: 0.9 ; Release Date: 91/10/08 .

SEE ALSO

co (1), ident(1), rcs (1), rcsdiff (1), rcsmerge (1), rlog (1), rcsfile (5), sccstorcs (1).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

NAME

clear – clear terminal screen

SYNOPSIS

clear

DESCRIPTION

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

FILES

/etc/termcap terminal capability data base

NAME

cmp – compare two files

SYNOPSIS

cmp [**–l** | **–x**] [**–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.

–x Print the byte number (decimal) and the differing bytes (hexadecimal) for each difference.

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

SEE ALSO

diff(1), comm(1)

DIAGNOSTICS

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

NAME

co – check out RCS revisions

SYNOPSIS

co [options] file ...

DESCRIPTION

Co retrieves revisions from RCS files. Each file name ending in ',v' is taken to be an RCS file. All other files are assumed to be working files. *Co* retrieves a revision from each RCS file and stores it into the corresponding working file.

Pairs of RCS files and working files may be specified in 3 ways (see also the example section).

1) Both the RCS file and the working file are given. The RCS file name is of the form *path1/workfile*,v and the working file name is of the form *path2/workfile*, where *path1/* and *path2/* are (possibly different or empty) paths and *workfile* is a file name.

2) Only the RCS file is given. Then the working file is created in the current directory and its name is derived from the name of the RCS file by removing *path1/* and the suffix ',v'.

3) Only the working file is given. Then the name of the RCS file is derived from the name of the working file by removing *path2/* and appending the suffix ',v'.

If the RCS file is omitted or specified without a path, then *co* looks for the RCS file first in the directory ./RCS and then in the current directory.

Revisions of an RCS file may be checked out locked or unlocked. Locking a revision prevents overlapping updates. A revision checked out for reading or processing (e.g., compiling) need not be locked. A revision checked out for editing and later checkin must normally be locked. Locking a revision currently locked by another user fails. (A lock may be broken with the *rcs* (1) command.) *Co* with locking requires the caller to be on the access list of the RCS file, unless he is the owner of the file or the superuser, or the access list is empty. *Co* without locking is not subject to accesslist restrictions.

A revision is selected by number, checkin date/time, author, or state. If none of these options are specified, the latest revision on the trunk is retrieved. When the options are applied in combination, the latest revision that satisfies all of them is retrieved. The options for date/time, author, and state retrieve a revision on the *selected branch*. The selected branch is either derived from the revision number (if given), or is the highest branch on the trunk. A revision number may be attached to one of the options **-l**, **-p**, **-q**, **-f**, or **-r**.

A *co* command applied to an RCS file with no revisions creates a zero-length file. *Co* always performs keyword substitution (see below).

–**l**[*rev*] locks the checked out revision for the caller. If omitted, the checked out revision is not locked. See option **-r** for handling of the revision number *rev*.

–**p**[*rev*] prints the retrieved revision on the std. output rather than storing it in the working file. This option is useful when *co* is part of a pipe.

–**q**[*rev*] quiet mode; diagnostics are not printed.

–**f**[*rev*] forces checkout of the specified revision regardless of whether a copy exists in the current directory.

–**d***date* retrieves the latest revision on the selected branch whose checkin date/time is less than or equal to *date*. The date and time may be given in free format and are converted to local time. Examples of formats for *date*:

22-April-1982, 17:20-CDT,
2:25 AM, Dec. 29, 1983,
Tue-PDT, 1981, 4pm Jul 21 (free format),
Fri, April 16 15:52:25 EST 1982 (output of ctime).

Most fields in the date and time may be defaulted. *Co* determines the defaults in the order year, month, day, hour, minute, and second (most to least significant). At least one of these fields must be provided. For omitted fields that are of higher significance than the highest provided field, the current values are assumed. For all other omitted

fields, the lowest possible values are assumed. For example, the date "20, 10:30" defaults to 10:30:00 of the 20th of the current month and current year. The date/time must be quoted if it contains spaces.

-r[*rev*] retrieves the latest revision whose number is less than or equal to *rev*. If *rev* indicates a branch rather than a revision, the latest revision on that branch is retrieved. *Rev* is composed of one or more numeric or symbolic fields separated by '.'. The numeric equivalent of a symbolic field is specified with the **-n** option of the commands *ci* and *rcs*.

-s*state* retrieves the latest revision on the selected branch whose state is set to *state*.

-w[*login*] retrieves the latest revision on the selected branch which was checked in by the user with login name *login*. If the argument *login* is omitted, the callers login is assumed.

-j*joinlist* generates a new revision which is the join of the revisions on *joinlist*. *Joinlist* is a comma-separated list of pairs of the form *rev2:rev3*, where *rev2* and *rev3* are (symbolic or numeric) revision numbers. For the initial such pair, *rev1* denotes the revision selected by the options **-l**, ..., **-w**. For all other pairs, *rev1* denotes the revision generated by the previous pair. (Thus, the output of one join becomes the input to the next.)

For each pair, *co* joins revisions *rev1* and *rev3* with respect to *rev2*. This means that all changes that transform *rev2* into *rev1* are applied to a copy of *rev3*. This is particularly useful if *rev1* and *rev3* are the ends of two branches that have *rev2* as a common ancestor. If *rev1* < *rev2* < *rev3* on the same branch, joining generates a new revision which is like *rev3*, but with all changes that lead from *rev1* to *rev2* undone. If changes from *rev2* to *rev1* overlap with changes from *rev2* to *rev3*, *co* prints a warning and includes the overlapping sections, delimited by the lines <<<<<<< *rev1*, =======, and >>>>>>> *rev3*.

For the initial pair, *rev2* may be omitted. The default is the common ancestor. If any of the arguments indicate branches, the latest revisions on those branches are assumed. If the option **-l** is present, the initial *rev1* is locked.

KEYWORD SUBSTITUTION

Strings of the form $*keyword*$ and $*keyword:...*$ embedded in the text are replaced with strings of the form $*keyword: value* $, where *keyword* and *value* are pairs listed below. Keywords may be embedded in literal strings or comments to identify a revision.

Initially, the user enters strings of the form $*keyword*$. On checkout, *co* replaces these strings with strings of the form $*keyword: value* $. If a revision containing strings of the latter form is checked back in, the value fields will be replaced during the next checkout. Thus, the keyword values are automatically updated on checkout.

Keywords and their corresponding values:

$Author$ The login name of the user who checked in the revision.

$CHeader$ This Convex extension is a reduced header containing the working file name, the revision number, and the date.

$Date$ The date and time the revision was checked in.

$Header$ A standard header containing the full pathname of the RCS file name, the revision number, the date, the author, and the state.

Id Identical to Header but only prints out the name of the RCS file and not the full pathname.

$Locker$ The login name of the user who locked the revision (empty if not locked).

Log The log message supplied during checkin, preceded by a header containing the RCS file name, the revision number, the author, and the date. Existing log messages are NOT replaced. Instead, the new log message is inserted after $*Log:...*$. This is useful for accumulating a complete change log in a source file.

$Revision$ The revision number assigned to the revision.

$Source$	The full pathname of the RCS file.
$RCSfile$	The name of the RCS file.
$State$	The state assigned to the revision with *rcs -s* or *ci -s*.

DIAGNOSTICS

The RCS file name, the working file name, and the revision number retrieved are written to the diagnostic output. The exit status always refers to the last file checked out, and is 0 if the operation was successful, 1 otherwise.

EXAMPLES

Suppose the current directory contains a subdirectory 'RCS' with an RCS file 'io.c,v'. Then all of the following commands retrieve the latest revision from 'RCS/io.c,v' and store it into 'io.c'.

```
co io.c;    co RCS/io.c,v;    co io.c,v;
co io.c RCS/io.c,v;    co io.c io.c,v;
co RCS/io.c,v io.c;    co io.c,v io.c;
```

FILE MODES

The working file inherits the read and execute permissions from the RCS file. In addition, the owner write permission is turned on, unless the file is checked out unlocked and locking is set to *strict* (see *rcs* (1)).

If a file with the name of the working file exists already and has write permission, *co* aborts the checkout if **-q** is given, or asks whether to abort if **-q** is not given. If the existing working file is not writable, it is deleted before the checkout.

FILES

The caller of the command must have write permission in the working directory, read permission for the RCS file, and either read permission (for reading) or read/write permission (for locking) in the directory which contains the RCS file.

A number of temporary files are created. A semaphore file is created in the directory of the RCS file to prevent simultaneous update.

IDENTIFICATION

Author: Walter F. Tichy
Revision Number: 0.11 ; Release Date: 91/10/08 .

SEE ALSO

ci (1), ident (1), rcs (1), rcsdiff(1), rcsmerge (1), rlog (1), rcsfile (5), sccstorcs (1).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

LIMITATIONS

The option **-d** gets confused in some circumstances, and accepts no date before 1970. There is no way to suppress the expansion of keywords, except by writing them differently. In nroff and troff, this is done by embedding the null-character '\&' into the keyword.

BUGS

The option **-j** does not work for files that contain lines with a single '.'.

NAME

col – filter reverse line feeds

SYNOPSIS

col [**–bfh**]

DESCRIPTION

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

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

If the **–b** option is given, *col* assumes that the output device in use is not capable of backspacing. In this case, if several characters are to appear in the same place, only the last one read will be taken.

The control characters SO (ASCII code 017), and SI (016) are assumed to start and end text in an alternate character set. The character set (primary or alternate) associated with each printing character read is remembered; on output, SO and SI characters are generated where necessary to maintain the correct treatment of each character.

If the **-h** option is given, *col* will convert white space to tabs to shorten printing time.

All control characters are removed from the input except space, backspace, tab, return, newline, ESC (033) followed by one of 7, 8, 9, SI, SO, and VT (013). This last character is an alternate form of full reverse line feed, for compatibility with some other hardware conventions. All other non-printing characters are ignored.

SEE ALSO

troff(1)(optional product), tbl(1)

BUGS

Can't back up more than 128 lines.
No more than 800 characters, including backspaces, on a line.

NAME

colcrt – filter nroff output for CRT previewing

SYNOPSIS

colcrt [–] [**–2**] [file ...]

DESCRIPTION

Colcrt provides virtual half-line and reverse line feed sequences for terminals without such capability, and on which overstriking is destructive. Half-line characters and underlining (changed to dashing '–') are placed on new lines in between the normal output lines.

The optional – suppresses all underlining. It is especially useful for previewing *allboxed* tables from *tbl*(1).

The option **–2** causes all half-lines to be printed, effectively double spacing the output. Normally, a minimal space output format is used which will suppress empty lines. The program never suppresses two consecutive empty lines, however. The **–2** option is useful for sending output to the line printer when the output contains superscripts and subscripts which would otherwise be invisible.

A typical use of *colcrt* would be

```
tbl exum2.n | nroff –ms | colcrt – | more
```

SEE ALSO

nroff(1), col(1), more(1), ul(1)

BUGS

Should fold underlines onto blanks even with the '–' option so that a true underline character would show; if we did this, however, *colcrt* wouldn't get rid of *cu'd* underlining completely.

Can't back up more than 102 lines.

General overstriking is lost; as a special case '|' overstruck with '–' or underline becomes '+'.

Lines are trimmed to 132 characters.

Some provision should be made for processing superscripts and subscripts in documents which are already double-spaced.

NAME

colrm – remove columns from a file

SYNOPSIS

colrm [startcol [endcol]]

DESCRIPTION

Colrm removes selected columns from a file. Input is taken from standard input. Output is sent to standard output.

If called with one parameter the columns of each line will be removed starting with the specified column. If called with two parameters the columns from the first column to the last column will be removed.

Column numbering starts with column 1.

SEE ALSO

expand(1)

AUTHOR

Jeff Schriebman

NAME

comm – select or reject lines common to two sorted files

SYNOPSIS

comm [– [**123**]] file1 file2

DESCRIPTION

Comm reads *file1* and *file2*, which should be ordered in ASCII collating sequence, and produces a three column output: lines only in *file1;* lines only in *file2;* and lines in both files. The filename '–' means the standard input.

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

SEE ALSO

cmp(1), diff(1), uniq(1)

NAME

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

SYNOPSIS

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 online 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 newly-compacted file has modification and access times matching the uncompacted file.

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.* The newly-uncompacted file has modification and access times matching the compacted file. Thus if a file is compacted, not accessed, and then uncompacted the modification and access times of the file will not change. If no file names are given to *uncompact* 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 252 characters to allow space for the appended '.C'.

FILES

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

NOTES

compact and *uncompact* are capable of processing files greater than two gigabytes in size.

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.

AUTHOR

Colin L. Mc Master

NAME

contact – submit a CONVEX problem report

SYNOPSIS

contact [-a entry] [-r]

DESCRIPTION

contact submits problem reports to the CONVEX Technical Assistance Center (TAC). The user will be queried for all of the necessary problem information, and then the report will be sent by electronic mail to CONVEX. An acknowledgement of receipt should be received by the submitter via electronic mail within 48 hours after the problem report was sent.

The types of information that are requested are explained below.

User-Information

The user's name, title, phone number, corporate name, and any other helpful identifying information about the site.

If a file by the name of ".contact" exists in the user's home directory, the contents of it will be included as the User-Information, and the user will not be prompted for it. A typical *.contact* file might look like:

Joe User
Software Engineer
Big Computer Corporation
(907) 555-1212

Product

The name of the product involved.

Version

The version number of the product involved.

If the product is one of the standard utilities, it's version may be displayed with the *vers*(1) command.

Summary

A one line summary of the problem. This summary should be as descriptive of the problem as possible (within the one line constraint), since it will be used in any correspondence about the problem report.

Description

A detailed description of the problem.

Priority

A measure of the importance of the problem. A list of problem priorities/explanations is displayed, from which one may be selected.

Repeat-By

Instructions on how to repeat the problem. This includes not only the commands necessary to reproduce the problem, but any flags used during compilations, descriptions of what was being attempted, etc.

Comments

Anything else that might be relevant to the problem report.

Documentation-Comments:

Suggestions or comments on the documentation you referenced when trying to resolve the problem (for example, organization, accessibility, corrections, additions).

Files

Which (if any) files are needed in order to reproduce the problem. If all of the files contain text and are small enough, they will be included in the report. If any of the files do not contain text and are not too big, they will be transmitted independently of the report to CONVEX by *uusend*. If any of the files are too big, instructions on what should be done will be displayed.

Only individual files may be named within this section. If an entire directory is needed, it should be either combined into a single file using a utility such as *tar*(1), or each file in the directory should be entered separately, one per line.

The sections User-Information, Description, Repeat-By, Comments, Documentation-Comments, and Files are "multi-line" sections. The other sections are "single-line" sections. Single-line sections are terminated by a <newline>. This means that only one line of information may be entered. Multi-line sections are terminated either by a single period (".") on a line, or by EOF (usually <control-d>). This was patterned after *mail*(1).

Lines beginning with a tilde ("~") are treated as special in multi-line sections. The character after the tilde is considered as a request to *contact* for some special function - except in the Files section, where it will be processed as a home directory expansion as in *csh*(1). Here is a summary of these tilde functions.

~e	Invoke a text editor on the information in this section. The editor invoked is the one defined by the **EDITOR** environment variable, or *vi*(1) by default.
~h	Display a list of the available tilde escapes.
~p	Print the information in this section on the terminal.
~r file	Read the contents of "file" into the current section.
~~	Insert a single tilde as the first character in the line.

Once all of the sections have been entered, the user is prompted for disposition of the problem report. There are four operations that may be performed on a finished problem report.

Review

The problem report may be reviewed with *more*(1). The user is once again prompted for report disposition.

Edit

A text editor is invoked on the problem report. The editor invoked is determined by the same method as described above. The user is once again prompted for report disposition.

Submit

The problem report may be submitted. This terminates the session with *contact*.

Abort

The problem report may be aborted. Selecting this option will force the contents of the report to be appended to "dead.report", in the user's home directory and *contact* will terminate. Invoking *contact* with the "**-r**" option will start *contact* up at this point with "dead.report" as input.

contact may be aborted at any time by using the interrupt key (usually <control-c>). Note that this does not save the contents of the report, as does the abort option described above.

A number of parameters used by *contact* are defined in the *contactcap*(5) configuration file. *contact* currently uses the only entry in the file, "c0", for these parameters. The "**-a**" option can be used to specify an alternate configuration file entry, should one be available.

contact returns an exit status describing what it did. The codes are defined in <*sysexits.h*>.

EX_OK	Successful completion.
EX_CANTCREAT	Could not create temporary file.
EX_NOINPUT	Necessary input file could not be accessed.
EX_TEMPFAIL	Temporary failure (aborted), try again.
EX_UNAVAILABLE	Mail/UUCP service to CONVEX does not exist.
EX_USAGE	Could not process a specified command line option.

FILES

/tmp/tac_*	working files
~/.contact	user information
~/dead.report	aborted report data
/usr/lib/contactcap	configuration file

SEE ALSO

csh(1), mail(1), more (1), uucp(1C), vers(1), contactcap(5)

RESTRICTIONS

Contact cannot read files with a protection mask of 750.

Files that are sent independently of the contact report by ***uusend*** are subject to the restriction that all systems along the line must have the ***uusend*** command available and allow the remote execution of it.

NAME

cp – copy

SYNOPSIS

cp [**–ip**] file1 file2

cp [**–ipr**] file ... directory

cp –r [**–ip**] directory1 directory2

DESCRIPTION

file1 is copied onto *file2*. By default, the mode and owner of *file2* are preserved if it already existed; otherwise the mode of the source file modified by the current *umask*(2) is used. The **–p** option causes *cp* to attempt to preserve (duplicate) in its copies the modification times and modes of the source files, ignoring the present *umask*.

In the second form, one or more *files* are copied into the *directory* with their original filenames. The *directory* must already exist.

The third form will recursively copy *directory1* to *directory2*. If necessary, *directory2* will be created.

cp refuses to copy a file onto itself.

If the **–i** option is specified, *cp* will prompt the user with the name of the file whenever the copy will cause an old file to be overwritten. An answer of "y" will cause *cp* to continue. Any other answer will prevent it from overwriting the file.

If the **–r** option is specified and any of the source files are directories, *cp* copies each subtree rooted at that name; in this case the destination must be a directory.

NOTES

cp is capable of copying files which are greater than two gigabyes in size. It may not be fast, but it can do it.

SEE ALSO

cat(1), mv(1), rcp(1C)

BUGS

On very large directory structures, *cp –r* may fail in the event that the recursion becomes too deep and the program runs out of stack space. A possible workaround for this case is to use tar:

cd *srcdir*; **tar cf - . | (cd** *destdir*; **tar xfB -)**

NAME

cpall – copy directory

SYNOPSIS

cpall sourcedir destdir

DESCRIPTION

Cpall recursively copies all files and directories from directory *sourcedir* to directory *destdir*.

SEE ALSO

cp(1), mkdir(1), mv(1), tar(1)

NAME

cpio – copy file archives in and out

SYNOPSIS

cpio –o[Bcv]
cpio –i[Bacdfmrtuv] [*pattern...*]
cpio –p[dlmruv] *directory*

DESCRIPTION

The **cpio** utility produces and reads files in the format specified by the **cpio Archive/Interchange File Format** specified in *IEEE Std. 1003.1-1988*.

The **cpio -i** (copy in) utility 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. Multiple *patterns* may be specified and if no *patterns* are specified, the default for *patterns* is selecting all files. The extracted files are conditionally created and copied into the current directory, and possibly any levels below, based upon the options described below and the permissions of the files will be those of the previous **cpio -o** . The owner and group of the files will be that of the current user unless the user has appropriate privileges, which causes **cpio** to retains the owner and group of the files of the previous **cpio -o** .

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

If an error is detected, the cause is reported and the **cpio** utility will continue to copy other files. **cpio** will skip over any unrecognized files which it encounters in the archive.

The following restrictions apply to the **cpio** utility:

1 Pathnames are restricted to 256 characters.

2 Appropriate privileges are required to copy special files.

3 Blocks are reported in 512-byte quantities.

Options

The following options are available:

–B Input/output is to be blocked 5120 bytes to the record. Can only be used with **cpio -o** or **cpio -i** for data that is directed to or from character special files.

–a Reset access times of input files after they have been copied. When the **–l** option is also specified, the linked files do not have their access times reset. Can only be used with **cpio -i** .

–c Write header information in ASCII character for portability. Can only be used with **cpio -i** or **cpio -o** . Note that this option should always be used to write portable files.

–d Creates directories as needed. Can only be used with **cpio -i** or **cpio -p** .

–f Copy in all files except those in *patterns* . Can only be used with **cpio -i** .

–l Whenever possible, link files rather than copying them. Can only be used with **cpio -p** .

–m Retain previous modification times. This option is ineffective on directories that are being copied. Can only be used with **cpio -i** or **cpio -p** .

–r Interactively rename files. The user is asked whether to rename *pattern* each invocation. Read and write permissions for **/dev/tty** are required for this option. If the user types a null line, the file is skipped. Should only be used with **cpio -i** or **cpio -o** .

–t Print a table of contents of the input. No files are created. Can only be used with **cpio -i** .

–u Copy files unconditionally; usually an older file will not replace a new file with the same name. Can only be used with **cpio -i** or **cpio -p** .

–v Verbose: cause the names of the affected files to be printed. Can only be used with **cpio -i** . Provides a detailed listing when used with the **–t** option.

Operands
The following operands are available:

patterns Simple regular expressions given in the name-generating notation of the shell.

directory The destination directory.

Exit Status
The **cpio** utility exits with one of the following values:

0 All input files were copied.

2 The utility encountered errors in copying or accessing files or directories. An error will be reported for nonexistent files or directories, or permissions that do not allow the user to access the source or target files.

It is important to use the **-depth** option of the **find** utility to generate pathnames for **cpio** . This eliminates problems **cpio** could have trying to create files under read-only directories.

The following command:

ls | cpio -o > ../newfile

copies out the files listed by the **ls** utility and redirects them to the file **newfile** .

The following command:

cat newfile | cpio -id "memo/al" "memo/b*"

uses the output file **newfile** from the **cpio -o** utility, takes those files that match the patterns **memo/al** and **memo/b*** , creates the directories below the current directory, and places the files in the appropriate directories.

The command

find . -print | cpio -pdlmv newdir

takes the file names piped to it from the **find** utility and copies or links those files to another directory named **newdir** , while retaining the modification time.

FILES

/dev/tty used to prompt the user for information when the **–i** or **–r** options are specified.

NOTES

cpio is capable of archiving and generating files of greater than two gigabytes in size. There is an upper limit of eight gigabytes on the size of a file that *cpio* can archive. There is no limit on the size of the final archive.

Redirection of *cpio* output from the command line will not allow the user to create archives greater than two gigabytes. The user will need to use a pipe line to create archives greater than two gigabutes in size. An example pipeline is

find . -print | cpio -oc | dd of=archive

which will write the final *cpio* archive into **archive**.

SEE ALSO

find(1), pax(1), tar(1), cpio(5), tar(5)

COPYRIGHT

AUTHOR

Mark H. Colburn
NAPS International
117 Mackubin Street, Suite 1
St. Paul, MN 55102
mark@jhereg.MN.ORG

Sponsored by **The USENIX Association** for public distribution.

NAME

cpp – C preprocessor

SYNOPSIS

/lib/cpp [*option*] ... [*infile* [*outfile*]]

DESCRIPTION

cpp is a macro preprocessor similar to the preprocessor used by the CONVEX C compiler *cc*(1). *cpp* is an ANSI C compatible preprocessor that can also be made compatible with previous versions of *cpp* with the *-pcc* option (described below).

COMPATIBILITY

There are four command line options that cause the preprocessor to conform with the ANSI C standard in varying degrees. These compatibility mode options are the same options that the C compiler uses.

–ext This mode causes the preprocessor to conform to an extended implementation of the ANSI C standard. This is the default mode. This is the extended mode.

–pcc This option causes the preprocessor to behave compatibly with preprocessors that existed prior to the ANSI C standard. This is the backward-compatible mode.

–std This option causes the preprocessor to conform to the ANSI C standard. This is the standard, conforming mode.

–str This option causes the preprocessor to strictly conform to the ANSI C standard. This is the strict, conforming mode.

COMMAND LINE OPTIONS

If no files are specified *cpp* reads lines from *stdin*, processes them, and writes them to *stdout*. *infile* and *outfile* may be specified to change this. There is no way to specify an output file without specifying an input file.

The following command line options are interpreted by *cpp*:

–B*dir* Finds substitute error message file, *errmsgc*, in the directory *dir*. If *dir* is not specified, the standard backup version in the directory /usr/lib/oldcc is used instead. The error message file is version specific.

–C Retains comments in the output. C style comments are usually removed.

–D*name*
–D*name*=*def*
Defines the *name* (as if by *#define*) as *def*, or 1 if *def* is omitted.

–I*dir* Adds *dir* to the search list for *#include* files.

–k Runs the preprocessor on the named C source file and generates a dependency description for make(1); results are printed out on the standard output stream. This option cannot process input from *stdin*.

–P Do not insert control lines into the output.

–U*name* Remove any initial definition of *name*. Names predefined by the preprocessor are discussed below.

–vn Identifies version. Outputs the version number of *cpp* to *stderr*.

Lines in the input file that have a "#" character as the first nonblank character are interpreted as directives by *cpp*. The effect of a directive lasts until the end of the source or until it is countered by another directive.

ENVIRONMENT VARIABLES

The CONVEX C preprocessor *cpp* prepends the value of the environment variable **CPPOPTIONS** (if it is set) to each command line so that options need not be specified every time *cpp* is invoked. For example,

```
setenv CPPOPTIONS '-pcc'
```

causes all preprocessing to occur in the backward-compatible mode.

The CONVEX C compiler and *lint* remove the value of **CPPOPTIONS** before invoking the preprocessor to avoid conflict. **CPPOPTIONS** is useful when invoking *cpp* as a general purpose preprocessor.

PREDEFINED NAMES

The following macros are predefined when you invoke *cpp*.

__convex__
This symbol is predefined in all compatibility modes as 1. To remain compatible with previous preprocessors, the "convex" symbol is also predefined only in the backward-compatible mode, but its availability may be discontinued in a future release of the preprocessor.

__DATE__
This symbol is predefined in the extended, standard, and strict modes of the preprocessor. It returns the current date. This definition may not be removed with the #undef directive or the *-U* command line option.

__FILE__
This symbol is predefined in all modes of the preprocessor. It returns the current file name. This definition cannot be removed with the #undef directive or the *-U* command line option.

__LINE__
This symbol is predefined in all modes of the preprocessor. It returns the current line number. This definition cannot be removed with the #undef directive or the *-U* command line option.

__STDC__
This symbol is predefined in the standard and strict modes of the preprocessor as the constant 1. This definition cannot be removed with the #undef directive or the *-U* command line option.

__stdc__
This symbol is predefined in the extended, standard, and strict modes of the preprocessor as the constant 1.

__TIME__
This symbol is predefined in the extended, standard, and strict modes of the preprocessor. It returns the current time. This definition may not be removed with the #undef directive or the *-U* command line option.

__unix__
This symbol is always predefined as constant 1. To remain compatible with previous preprocessors, the "unix" symbol is also predefined only in the backward-compatible mode, but its availability may be discontinued in a future release of the preprocessor.

When invoked by the CONVEX C compiler other names may be predefined. See *cc*(1) for details.

Other names beginning with "__" may be predefined by *cpp*. Such names are reserved to CONVEX; their usage or availability may change in subsequent releases. Applications should not depend on the *presence or absence* of names beginning with "__" (except those defined above).

TOKEN REPLACEMENT

A control line of the form:

#define *identifier replacement-string*

is a simple macro definition. It causes *cpp* to replace subsequent instances of the identifier with the replacement-string. A line of the form:

#define *identifier(argument, ...) replacement-string*

is a macro definition with arguments. Subsequent instances of the identifier are replaced by the

replacement-string in the definition. Each occurrence of an argument mentioned in the formal parameter list of the definition is replaced by the corresponding actual argument from the call.

The ## token-pasting operator, which may appear only in the replacement-string of #define directive, causes catenation of its operands. Thus

```
#define x a ## b
```

is equivalent to

```
#define x ab
```

The token-pasting operator is not available when *-pcc* is specified.

The # stringizing operator may only appear immediately before a formal parameter name in the replacement-string of a #define directive. When expansion of the macro occurs the actual parameter becomes a string constant. Thus

```
#define f(x)    #x[0]
f(hello)
```

produces

```
"hello"[0]
```

This operator is not available when *-pcc* is specified.

A long macro definition may be continued on another line by adding "\" at the end of the line to be continued.

The replacement string is rescanned for more defined identifiers.

A control line of the form:

#undef *identifier*

causes the identifier's macro definition to be deleted.

FILE INCLUSION

A control line of the form:

#include "*filename*"

or

#include <*filename*>

causes the replacement of that line by the entire contents of *filename*. Files included by a #include statement may contain further *#include* statements; include files may be nested.

The directory search order for *#include* files is (1) the directory of the file that contains the request unless the form **#include** <*filename*> was used (2) the directories specified by *-I*, and (3) the standard directory (*/usr/include*).

CONDITIONAL INSERTION

A control line of the form:

#if *constant-expression*

checks whether the constant expression evaluates to a nonzero quantity. A constant expression is any legitimate C expression that evaluates to a constant. See *"The C Programming Language"* for more information. A control line of the form:

#ifdef *identifier*

checks whether the identifier is currently defined in *cpp*. A control line of the form:

#ifndef *identifier*

checks whether the identifier is currently undefined in *cpp*.

Each form of the if statement may be followed by an arbitrary number of lines of text, optionally followed by

#elif *constant-expression*

and more lines of text. Multiple *#elif* constructs may be used.

After all the #elif lines a

#else

line may appear. The conditional input section is terminated by a

#endif

line.

The text is included or omitted according to the truth of the expression.

The expression on a #if or #elif line is subject to macro expansion.

These constructs may be nested.

LINE CONTROL

For other preprocessors that generate C programs, a line of the form:

#line *integer-constant [string-constant]*

causes the C compiler to believe, for purposes of error diagnostics, that the number of the next source line is given by the integer-constant and the current input file is given by the string-constant. If the string-constant is absent, the current filename does not change.

COMMENTS

C comments are replaced by a blank unless the *-C* option or the *-pcc* option is specified.

When the *-C* option is specified comments are retained in the output.

When the *-pcc* option is specified (without *-C*) comments are removed from the text; no blank is introduced in place of the comment. This feature is often used to catenate two pieces of text; such usage is not portable. Thus, in "*foo/* comment */bar*", the output will be "foobar" unless either "foo" or "bar" is a macro name. Expansion of "foo" and "bar" will take place if they are macro names. No macro expansion of "foobar" will be performed when it does not occur in a macro definition.

OTHER CONSIDERATIONS

The following features are not portable and are supported only with the *-pcc* flag.

Formal macro parameters are recognized in *# define* token strings, even inside character constants and quoted strings. The output from:

```
#define foo(a) '\a'
foo(bar)
```

is the six characters `'\bar'`.

Macro names are not recognized inside character constants or quoted strings during the regular scan. Thus:

```
#define foo bar
printf("foo");
```

does not expand "foo" in the second line, because it is inside a quoted string that is not part of a *# define* or *# undef*.

A mismatch between the number of formals and actuals in a macro call produces only a warning, not an error. Excess actuals are ignored; missing actuals are turned into null strings.

SEE ALSO

cc(1)

CONVEX C Guide

American National Standard for Information Systems — Programming Language C Document Number: X3J11/90-013

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

B. W. Kernighan and D. M. Ritchie, *The C Programming Language, Second Edition*

NAME

cpr - print "C" files

SYNOPSIS

cpr [**-l** *pagelength*] [**-o**] [**-f** *name*] [**-n**] [**-t**] [-] [*file*] ...

DESCRIPTION

cpr produces a formatted listing of one or more *files*, preceded by a table of contents. The output is separated into pages headed by the name of the file (in **boldface**), a date, and the page number. The date is the last modified time of the file or the current date if printing the standard input. The standard input is read if no files are given, or if a name of "-" (dash) is given as a file argument. Each file is scanned for C function declarations and terminating "}" (brace) characters. The function names found are added to the table of contents and are double stricken in the output listing. By default, blank space is inserted into the output after each closing brace. This has the effect of isolating function and structure declarations in the output.

The following options are allowed, and may be given in any order:

-l*pagelength*
: The following argument is a number to be used as the length of the page in lines, instead of the default 66.

-o Cause function name part of the line to be double stricken.

-f*name*
: Only the named function will be listed.

-n Cause the source lines to be sequentially numbered.

-t Only print the table of contents.

FILES

/tmp/cpr$$	temp file holding text
/tmp/CPR$$	temp file holding table of contents

SEE ALSO

cat(1), pr(1)

DIAGNOSTICS

Complains if it can't open any file.
Complains if it can't execute */bin/cat*.
Complains if the table of contents overflows (it will be truncated).

NAME

cref – cross reference program

SYNOPSIS

cref [*-n*] [**-t**] [**-l**] file...

DESCRIPTION

Cref generates a complete cross reference listing of one or more C programs, printing the result on the standard output. *Cref* prints cross reference listing containing all the programs' symbols alphabetically arranged, one to a line, with each line containing the numbers of the lines in the programs where the symbol was referenced. If the symbol was defined on a given line, that line number will be followed by a '#'. Symbols with more than approximately 15 references occupy multiple lines. There is no limit on the number of symbols that *cref* will handle, nor on the number of references per symbol.

Cref stores its symbols in a hash table whose size is determined by *cref* based on the total number of characters in the files to be processed. For almost all programs, this turns out to be an excellent approximation. However, for a few programs, generally short header files, there may be too many symbols for the hash table, and the diagnostic "Hash table overflowed!" will be printed out. If this occurs, *cref* should be rerun with the **-n** option, where **n** is some number. This will multiply the starting size of the hash table by **n** times.

If *cref* is invoked with the **-t** option, instead of its regular output it produces an output identical in form to that produced by the *ctags(1)* program. The advantage of the *cref* output over *ctags* is that *cref* will flag all variable and macro definitions as well as all function definitions.

If the **-l** option is used, a numbered source file listing precedes the cross reference listing.

SEE ALSO

ctags(1)

BUGS

Cref occasionally flags a reference as a definition when it really isn't. This most frequently happens after a **struct.**

NAME

creset - controller reset utility

SYNOPSIS

creset *ccu_number bus_number interrupt_level*

DESCRIPTION

creset is a program that runs on the SPU and is used to reset controller boards.

creset is invoked with the following arguments:

CCU_number	The CCU number to which the controller is attached.
bus_number	The bus number to which the controller is attached.
interrupt_level	The interrupt level of the controller.

SEE ALSO

cdown(1), cdump(1)

NOTES

creset is not supported by all controllers, using the utility to reset a controller which does not support *creset* will result in an error. Consult the specific product documentation to determine if *creset* is supported, as well as for the procedure in which it should be used.

NAME

cron – user clock daemon

SYNOPSIS

/etc/cron [–m]

DESCRIPTION

cron executes commands at specified dates and times according to the instructions found in *.crontab* files. 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.local*; see *init*(8).

A *.crontab* file may reside in any user's home directory. Once an hour, or when manually forced (see *tellcron*(1)), *cron* checks all users to see if any one has created or modified a *.crontab* file. For each user that has, the user's old commands (if any) are removed from *cron*'s event list, and his new commands are added.

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

minute (0-59),
hour (0-23),
day of the month (1-31),
month of the year (1-12),
and day of the week (1-7 with 1=Monday).

Each of these patterns may be either an asterisk (meaning all legal values), or a list of elements. A list of elements is either an element, or a set of elements where the elements are separated by commas. An element is either a number in the specified range, or two numbers separated by a minus sign (indicating an inclusive range). The sixth field is a string that is executed by the Shell at the specified times. A percent character in this field 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.

If a file exists in the user's home directory named *.crontab.host*, where *host* is the name reported by *hostname*(1), this file will be honored instead of the normal *.crontab* file. The facility allows a home directory to be mounted on more than one machine without forcing all commands to be run on all machines.

If both the day of the month and the day of the week fields are specified, the values are logically ANDed together. That is, if the entry lists Monday and Friday as the day of the week element, and the entry lists the 15th as the day of the month element, the command will only be executed if the 15th of the month falls on a Monday or Friday.

If a *.cronrc* file exists in a user's home directory, this file will be used to set the environment for each of his commands. A *.cronrc* file consists of lines that set environment variables (e.g., **PATH=:/bin:/usr/ucb:/usr/bin**). *cron* automatically sets the USER, HOME and SHELL variables for the user. Unless otherwise specified in the *.cronrc* file, *cron* will set **PATH=:/usr/local/bin:/usr/ucb:/bin:/usr/bin:/usr/convex** as the default search path, and SHELL to the user's default login shell. Valid shells are */bin/sh*, */bin/csh*, */bin/oldcsh*, */bin/ncsh*, */bin/tcsh*, */bin/ksh*, and */usr/convex/covue*. If a shell other than one of these is specified in *.cronrc*, commands will not be executed and an error message will be mailed to the user.

If a user has the variable CRONREPORT set in their *.cronrc* file, *cron* will report to them via mail any normal or error output generated by that user's commands, failure to find a command, or an abnormal exit status by a command. The exit status is as returned by the *wait*(2) system call, so its value is shifted 8 bits to the left. The system manager can enable this on a system-wide basis by invoking *cron* with the **–m** option out of *rc.local*. If CRONREPORT is not set, or **–m** has not been used, all output generated by user commands is discarded.

If the user is the superuser, *root*, *cron* uses */bin/sh* as the default shell. Other shells may be specified as usual by creating a *.cronrc* file in *root*'s home directory. *root*'s crontab file is */usr/lib/crontab*; a link exists from */.crontab* to */usr/lib/crontab* to maintain compatibility.

The file */usr/lib/crontab* must be owned by root. In all other cases, the user must own the *.crontab* and *.cronrc* files in his home directory. Furthermore, these files may not be writable by anyone but their owner. If these conditions are not all met, no commands will be executed for the

user, and a mail message will be sent to the user explaining the problem.

EXAMPLES

On the first day of every month, remove all of the user's ".o" files that have not been accessed for a week:

```
0 0 1 * * find ~ -name ".o" -atime +7 -exec rm {} \;
```

Run messages once an hour during business hours, Monday through Friday. Mail the standard error and standard output of the command to the user *george*.

```
0 9-17 * * 1-5 msgs -f |& mail george
```

Change directory to *~/bin* and run "make update" at 6 a.m. and 6 p.m. every day of the year. Direct the output of make to *~/bin/ERRS*:

```
0 6,18 * * * cd ~/bin; make update >& ERRS
```

Every Wednesday afternoon at 1:30 p.m. mail "Weekly staff meeting at 2:00" to the mail-alias called staff (the message is taken from the cron command line as standard input):

```
30 13 * * 3 mail staff%Weekly staff meeting at 2:00
```

FILES

$HOME/.crontab - file of *cron* commands
$HOME/.cronrc - *cron* environment
/etc/rc.local - file of site-dependent commands executed at system startup.
/usr/adm/cron.pid - process ID of cron daemon
/tmp/cronNNNNN - temporary file used for holding *stdin* to commands

SEE ALSO

tellcron(1), crontab(5), syslog(3)

WARNINGS

After changing a crontab, if you want *cron* to immediately use the new crontab, you must run *tellcron* manually.

Daylight savings time causes some interesting results from *cron*. Events scheduled to run between 2:00 AM and 3:00 AM will not be ran on the night that DST goes into effect, since those times never occur. Similarly, events scheduled to run between 1:00 AM and 2:00 AM will be ran twice on the night that DST ends, since those times occur twice.

Users in a networked environment should be aware that a personal *.crontab* file will be read by and acted upon by every machine which imports the directory in which that file resides. One way to ensure that commands in a personal *.crontab* file are run on only one machine is to place them in a shell script which tests the hostname of the machine on which it is running. That is, the commands are conditionally executed by the script, which is run by *cron*.

DIAGNOSTICS

All user-level *cron* diagnostics are mailed directly to the user. System-specific diagnostics are logged using the syslog(3) facility. A command will not be executed if it generates any diagnostic message.

A diagnostic is generated:

If a syntax error exists in the *.crontab* file. Syntax errors include too few fields in a command and invalid values in the time specifier fields.

If the *.crontab* file cannot be opened.

If the *.cronrc* file cannot be read.

If the user has more than 100 lines in the *.cronrc* file.

If the user specifies a shell unknown to *cron*.

If it cannot create a standard input file for a command in */tmp*.

cron is clever and can usually point out specific syntax errors.

Diagnostic mail messages will be repeated if the cron daemon is ever killed and restarted, i.e., after a system crash.

NOTES

To provide backwards compatibility, Sunday may be specified as a *0* or as a *7* in the day of the week field (1-7 with 1=Monday, 0=Sunday, and 7=Sunday).

NAME

crypt – encode/decode

SYNOPSIS

crypt [-s] [password]

DESCRIPTION

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

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

will print the clear.

The **-s** option enables secure mode.

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

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

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

Secure mode applies an additional level of hashing to the input before passing it to the Enigma machine.

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

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

FILES

/dev/tty for typed key

SEE ALSO

ed(1), makekey(8)

NOTES

Crypt (1) is not supported in the international distribution due to export restrictions.

NAME

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

SYNOPSIS

csh [**–cefinstvVxX**] [arg ...]

DESCRIPTION

csh is an implementation of a command language interpreter incorporating a history mechanism (see **History Substitutions**), job control facilities (see **Jobs**), interactive command line and history editing (see **Command Line Editing**), interactive command, file name, and user name completion (see **File Name Completion**) and a C-like syntax.

An instance of *csh* begins by executing commands from the file *.cshrc* in the home directory of the invoker. Typical items to include in a *.cshrc* file are aliases and variable settings. There are certain advantages to avoiding setting a lot of aliases or the prompt when the current shell is not interactive. This can be accomplished by the inclusion of the line

if (! $?prompt) exit

in your *.cshrc* file immediately after the lines that you want to have executed *every* time a shell is invoked. If this is a login shell, then *csh* also executes commands from the global login file */etc/login*, and from the home directory file *.login*, in that order. It is typical for crt users to put the command **stty crt** in their *.login* file and to invoke *tset*(1) there.

In the normal case, the shell then begins 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 first from the file *.logout* in the users home directory and then from the global logout file */etc/logout*.

Lexical structure

The shell splits input lines into words at blanks and tabs with the following exceptions. The characters &, |, ;, <, >, (, and) form separate words. If doubled in &&, ||, <<, or >>, these pairs form single words. These parser metacharacters may be made part of other words. To prevent their special meaning, precede 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 that continues to the end of the input line. This special meaning is prevented when the # is preceded by \ or is in quotations using `, ', or ".

Commands

A simple command is a sequence of words, the first of which specifies the command to be executed. A sequence of simple commands may be concatenated together, separated by ;. The commands will be executed sequentially. 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 that 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 id* with each command line. It keeps a table of current jobs, printed by the *jobs* command, and assigns them small integer numbers.

If you are running a job and wish to do something else, you may hit the keys ^**Z** (control-Z). This sends a STOP signal to the current job. The shell normally indicates that the job has been "Stopped" and prints 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 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. There is another special command, ^**Y**, which does not generate a STOP signal until a program attempts to *read*(2) it. This can usefully be typed ahead when you have prepared some commands for a job that you wish to stop after it has read them.

When a job is started in the background with &, the shell prints a line which looks like:

[1] 1234

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

A job being run in the background stops 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, background jobs will stop when they try to produce output, just as they do when they try to read input.

There are several ways to refer to jobs in the shell. The character % introduces a job name. If you wish to refer to job number 1, you can name it as **%1**. Just naming a job brings it to the foreground; thus **%1** is a synonym for **fg %1**, bringing job 1 back into the foreground. Similarly, entering **%1 &** resumes job 1 in 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**. This 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.

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

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 notifies you immediately of changes of status in background jobs. There is also a shell command *notify* that marks a single process so that its status changes are immediately reported. By default, *notify* marks the current process; simply say **notify** after starting a background job to mark it.

Command Line Editing

This shell provides a command line editor allowing the user to edit and correct their command line input. By default, the editor is an **emacs** style editor, but **vi** bindings are available. The bindings can be displayed by entering *bindkey* at the shell prompt.

Documentation lookup

The editor function *run-help* (Meta-h) prints a help file on the current command (using the same definition of current as the completion routines use). This help file is found by searching the path list **HPATH** for files of the form foo.help, foo.1, foo.8, or foo.6 in that order (assuming that the current command is foo). The file is just printed, not paged in any way. This is because *run-help* is meant to be used to look up short help files, not manual pages (although it can do manual pages also).

File name completion

In typing commands, it is not necessary to type a complete name, only a unique abbreviation is necessary. When you type a TAB to *csh* it will complete the name, echoing the full name on the terminal (and entering it into the edit buffer). If the prefix you typed matches no name, the terminal bell is rung, unless the variable *nobeep* is set. The name may be partially completed if the prefix matches several longer names. If this is the case, the name is extended up to the point of ambiguity, and the bell is rung. This works for file names, command names, shell variables and the (csh) ~ user name convention. The variable *fignore* may be set to a list of suffixes to be disregarded during completion.

Example

Assume the current directory contains the files:

DSC.TXT	bin	cmd	lib	memos
DSC.NEW	chaos	cmtest	mail	netnews
bench class	dev	mbox	new	

The command:

% gnumacs ch[TAB]

causes *csh* to complete the command with the file name chaos. If instead, the user types:

% gnumacs D[TAB]

csh extends the name to DSC and rings the terminal bell, indicating partial completion. However, if *fignore* had previously been set to a list containing .NEW as one element, e.g. (.o .NEW), *csh* would have completed the 'D' to DSC.TXT.

File name completion works equally well when other directories are addressed. Additionally, *csh* understands the C shell tilde (~) convention for home directories. Thus,

% cd ~speech/data/fr[TAB]

does what one might expect. This may also be used to expand login names only. Thus,

% cd ~co[TAB]

expands to

% cd ~convex

Command names may also be completed, for example,

% **gnum[TAB]**

will expand to "gnumacs" (assuming that there are no other commands that begin with "gnum").

Completion also works when the cursor is in the middle of the line, rather than just the end. All of the text after the cursor will be saved, the completion will work (possibly adding to the current name), and then the saved text will be restored in place, after the cursor.

The behavior of the completion can be changed by the setting of several shell variables:

Setting the **recexact** variable makes an exact command be expanded rather than just ringing the bell. For example, assume the current directory has two subdirectories called foo and food, then with **recexact** set the following could be done:

% **cd fo[TAB]**

to ...

% **cd foo[TAB]**

to ...

% **cd foo/**

rather than beeping on the second TAB.

With the **autolist** variable set, attempting completion when several choices are possible automatically lists the choices, effectively merging the functionality described in the next section into the completion mechanism. The "noise level" can be controlled by the value that **autolist** is set to: With **autolist=beepnone**, completion will only beep if there are no matching names; with **autolist=beepnever**, completion will never beep. Any other value (or no value) gives the default beeping behavior above.

When the **autoexpand** variable is set, the expand-history function is invoked automatically before the completion attempt, expanding normal csh history substitutions.

For covert operation, the variable **nobeep** can be set; it prevents the completion mechanism, as well as *csh* in general, from actually beeping.

Listing of possible names

At any point in typing a command, you may request "what names are available". Thus, when you have typed, perhaps:

% **cd ~speech/data/fritz/**

you may wish to know what files or subdirectories exist (in ~speech/data/fritz), without, of course, aborting the command you are typing. Typing the character Control-D (^D), will list the names (files, in this case) available. The files are listed in multicolumn format, sorted column-wise. Directories are indicated with a trailing '/', executable files with a '*', symbolic links with a '@', sockets with a '=', named pipes (FIFOs) with a '|', character devices with a '%', and block devices with a '#'. Once printed, the command is re-echoed for you to complete.

Additionally, one may want to know which files match a prefix. If the user types:

% **cd ~speech/data/fr[^D]**

all files and subdirectories whose prefix was λqfr" are printed. Notice that the example before was simply a degenerate case of this with a null trailing file name. (The null string is a prefix of all strings.) Notice also, that a trailing slash is required to pass to a new directory for both file name completion and listing.

The degenerate

% ~[^D]

will print a full list of login names on the current system. Note, however, that the degenerate

% <Spaces>[^D]

does not list all of the commands, but only beeps.

Listing/expanding of words that match a name containing wildcard characters can be done via the list-glob/expand-glob function:

```
% ls
foo.c  bar.c  a.out
% vi *.c[^X]g
foo.c bar.c
% vi *.c[^X]*
% vi foo.c bar.c
```

Command name recognition

Command name recognition and completion works in the same manner as file name recognition and completion above. The current value of the environment variable **PATH** is used in searching for the command. For example

% newa[TAB]

might expand to

% newaliases

Also,

% new[^D]

would list all commands (along **PATH**) that begin with "new".

Note that ^D has three different effects on *csh*. On an empty line (one that contains nothing, not even spaces), *^D* sends an EOF to *csh* just as it does for normal programs. When the cursor is in the middle of a line of text, *^D* deletes the character that the cursor is under. Finally, a *^D* at the end of a line of text lists the available names at that point. To get a list of available names when the cursor is in the middle of a line (or on an empty line), a Meta-Control-D should be typed (Escape followed by Control-D on most keyboards).

Spelling correction

If while typing a command, the user mistypes or misspells a file name, user name, or command name, *csh* can correct the spelling. When correcting a file name, each part of the path is individually checked and corrected. Spelling correction can be invoked in several different ways:

The *spell-word* function, normally bound to M-s (and M-S), will attempt to correct the word immediately before the cursor. For example, typing:

% cd /uxr/spol/news[ESC s]

csh will check the path for spelling, correct the mistakes, and redraw the line as

% cd /usr/spool/news

leaving the cursor at the end of the line.

Spelling correction of the entire command line (independent of where the cursor is) can be done with the *spell-line* function, normally bound to M-$ (Meta-Dollar-sign). It will check each word independently, but in order to avoid corrupting command options, no correction is attempted on words whose first character is found in the string "!.\^-*%".

Finally, automatic spelling correction will be done each time the Return key is hit, if the **correct** variable is set to an appropriate value: **correct=cmd** will cause the spelling of the command name only to be checked, while **correct=all** causes checking of all words on the line, like the spell-line function. If any part of the command line is corrected, the user will be given a special prompt as defined by the **prompt3** variable, followed by the corrected line, e.g.

```
% ls /usr/bin
CORRECT>ls /usr/bin (y/n)?
```

Answering 'y' or <space> at the prompt will cause the corrected line to be used, anything else will leave the original line unchanged.

Substitutions

The various transformations the shell performs on the input are described in the sections below 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 that contains history substitution is echoed on the terminal before it is executed as it could have been typed without history substitution.

Commands that are input from the terminal and that 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. Commands are numbered sequentially from 1. The size of the history list is controlled by the *history* variable. The previous command is always retained, regardless of its value.

Consider the following output from the *history* command:

```
 9  14:02  write michael
10  14:10  ex write.c
11  14:20  cat oldwrite.c
12  14:22  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 ! or **%h** in the prompt string.

With the current event (13) we can refer to previous events by event number **!11**, relatively (as in **!-2**, referring to the same event), by a prefix of a command word (as in **!d** for event 12, or **!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, follow the event specification by **:** and a designator for the desired words. The words of an 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 ^, $, *, –, or %. A sequence of modifiers, each preceded by **:**, can be placed after the optional word designator. 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 *r* for *l*
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., **!$**. In this case, the reference is to the previous command. However, if a previous history reference occurred on the same line, this form repeats the previous reference. Thus **!?foo? !$** gives the first and last arguments from the command matching **?foo?**.

The special history reference **!#** does not refer to any previous command, but refers to the current command. It can be used to duplicate any or all of the words just typed. For example, **ls /usr/src/lib !#^/libc** is equivalent to **ls /usr/src/lib /usr/src/lib/libc**.

A special abbreviation of a history reference occurs when the first nonblank character of an input line is a ^. This is equivalent to **!:s^** providing a convenient shorthand for substitutions on the text of the previous line. Thus, **^lb^lib** fixes the spelling of **lib** in the previous command.

Finally, a history substitution may be surrounded with **{** and **}**, if necessary, to insulate it from the characters that follow. Thus, after **ls –ld ~paul**, you might enter **!{l}a** to do **ls –ld ~paula**, while **!la** would look for a command starting with **la**.

Visual History

The keys ^P and ^N are used to step up and down the history list. If the user has typed in the following:

```
% ls
foo     bar
% echo mumble
mumble
%
```

then enters ^P, the shell will place "echo mumble" into the editing buffer, and will put the cursor at the end of the line. If another ^P is entered, then the editing line will change to "ls". More ^Ps will make the bell ring, since there are no more lines in the history. ^N works the same way, except it steps down (forward in time).

An easy way to re-do a command is to type ^P followed by <return>. Also, pieces of previous commands can be assembled to make a new command. The commands that work on regions are especially useful for this.

^P and ^N actually only copy commands from out of the history into the edit buffer; thus the user may step back into the history and then edit things, but those changes do not affect what is actually in *csh*'s history.

Another way to recall (parts of) history commands is via the 'expand-history' function. A variation of the 'expand-history' function is called 'magic-space'. This function expands csh history, and always appends a space. Magic-space thus can be bound to <space>, to automatically expand csh history. Expand-history is normally bound to *M-<space>* and magic-space is not bound.

Searching the Visual History

Two new editor functions have been added: history-search-backward, bound to M-p (and M-P), and history-search-forward, bound to M-n (and M-N). Each of these search backward (or forward) through the history list for previous (next) occurrence of the first word in the input buffer as a command. Entering:

```
% echo foo
foo
% ls
filea     fileb
% echo bar
bar
%
```

and then enters "echo<ESC>p", "echo bar" will be in the editing buffer. If another *M-p* was entered, the editing buffer would change to "echo foo". This capability is compatible with the plain visual history; if the user were to then enter ^P the editing buffer would be changed to "ls".

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 in **Variable Substitution**.

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

Thus, if the alias for **ls –l** is **ls**, the command **ls /usr** would map to **ls –l /usr**, the argument list here being undisturbed. Similarly, if **lookup** was an alias for **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 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 that *pr*'s its arguments to the line printer.

Variable substitution

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

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

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

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

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

Unless enclosed in " or given the **:q** modifier, the results of variable substitution may eventually be command and filename substituted. Within ", a variable whose value consists of multiple words expands to a portion of a single word, with the words of the variables value separated by blanks. When the **:q** modifier is applied to a substitution, the variable expands 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 that 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 that 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 (but : modifiers and the other forms given below are not available in this case).

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

$# name
${# name}
Gives the number of words in the variable. This is useful for later use in a *[selector]*.

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

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

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

The modifiers **:h**, **:t**, **:r**, **:q**, and **:x**, as described in **History Substitution** may be applied to the substitutions above, as may **:gh**, **:gt**, and **:gr**. If braces **{ }** appear in the command form, then the modifiers must appear within the braces. *The current implementation allows only one : modifier on each $ expansion.*

The following substitutions may not be modified with : modifiers.

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

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

$$
Substitutes 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 substitutions, are applied selectively to the arguments of builtin commands. This means that portions of expressions that are not evaluated are not subjected to these expansions. For commands that are not internal to the shell, the command name is substituted separately from the argument list.

Command substitution

Command substitution is indicated by a command enclosed in single quotation marks. 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 double quotation marks, only newlines force new words; blanks and tabs are preserved.

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

Filename substitution

If a word contains any of the characters *, ?, [, {, or 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 filenames that match the pattern. In a list of words specifying filename substitution, it is an error for no pattern to match an existing filename, but it is not required for each pattern to match. Only the metacharacters *, ?, and [imply pattern matching, the characters ~ and { are more akin to abbreviations.

In matching filenames, the . at the beginning of a filename or immediately following a /, as well as the /, must be matched explicitly. The * matches any string of characters, including the null string. The ? matches any single character. The sequence [...] acts identically to the regular expression usage in other utilites. It specifies a range of valid characters, unless the first character after the [(open bracket) is a ^ (caret), in which case, it specifies a list of characters not to match on. Within [...], a pair of characters separated by – specifies a list of all characters lexically between the listed two.

The ~ at the beginning of a filename is used to refer to home directories. Standing alone i.e., ~, it expands to the invoker's 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 the user's home directory. Thus, **~ken** might expand to **/usr/ken** and **~ken/chmach** to **/usr/ken/chmach**. If the ~ is followed by a character other than a letter or /, or appears other than at the beginning of a word, it is left undisturbed.

The metanotation **a{b,c,d}e** is 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.

Directory stack element access

csh will now allow the user to access all elements in the directory stack directly. The syntax "=<digit>" is recognized by csh as indicating a particular directory in the stack. (This works for the file/command name recognition as well.) This syntax is analogous to the ~ syntax for access to users' home directories. The stack is viewed as zero-based, i.e., =0 is the same as $cwd, which is the same as ".". As a special case, the string "=-" is recognized as indicating the last directory in the stack. Thus,

```
% dirs -v
0 /usr/net/bin
1 /usr/spool/uucp
2 /usr/accts/sys
% echo =2
/usr/accts/sys
% ls -l =1/LOGFILE
-rw-r--r-- 1 uucp    2594 Jan 19 09:09 /usr/spool/uucp/LOGFILE
```

```
% echo =-/.cs*
/usr/accts/sys/.cshrc
% echo =4
Not that many dir stack entries.
%
```

csh will complain if you ask for a directory stack item which does not exist.

Normally, the command "pushd +2" would rotate the entire stack around through 2 stack elements, placing the entry found there at the top of the stack. If, however, the new shell variable *dextract* is set, then issuing "pushd +n" will cause the nth directory stack element to be extracted from its current position, which will then be pushed onto the top of the stack. For example:

```
% dirs
~ /usr/spool/uucp /usr/net/bin /sys/src
% set dextract
% pushd +2
/usr/net/bin ~ /usr/spool/uucp /sys/src
% unset dextract
% pushd +2
/usr/spool/uucp /sys/src /usr/net/bin ~
```

Input/output Redirection

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 that is identical to *word*. *word* is not subjected to variable, filename, or command substitution, and each input line is compared to *word* before any substitutions are done on this input line. Unless a quoting \, ", ', or ` appears in *word*, variable and command substitution is performed on the intervening lines, allowing \ to quote $, \, and `. Commands that 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 that 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, it is truncated and its previous contents are lost.

  If the variable *noclobber* is set, then the file must not exist or be a character special file (e.g., a terminal or **/dev/null**) or an error results. This helps prevent accidental destruction of files. In this case, the ! forms can be used 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 that < 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 that run from a file of shell commands have no access to the text of the commands by default; rather, they receive the original standard input of the shell. The << mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block-read its input. Note that the default standard input for a command run detached 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 =~ is pattern equivalence and !~ is pattern non-equivalence much as != is numerical non-equivalence and == is numerical equivalence. Pattern (non)equivalence means the right side of the expression is a *pattern* (containing, e.g., *, ?, and instances of [...]) against which the left 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 that begin with **0** are considered octal numbers. Null or missing arguments are considered **0**. The result of all expressions are strings that 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 that are syntactically significant to the parser (&, |, <, >, (, and)), they should be surrounded by spaces.

Also available in expressions as primitive operands are command executions enclosed in { and } and file inquiries 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
l	symbolic link
p	named pipe (fifo)

The specified name is command and filename expanded, and then tested to see if it has the specified relationship to the real user. If the file does not exist or is inaccessible, then all inquiries 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.

Note that the –x test checks only the permission bits on the file in question (see *access (2)*. As a result, this test can be fooled into giving false information, especially in the cases of set-user/group-ID binaries in non-SUID NFS file systems, set-user/group-Id shell scripts, and generic ASCII files with their execute bits set.

Control flow

The shell contains a number of commands that 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 shells 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 gotos will succeed on nonseekable inputs.)

Builtin commands

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

alias
alias *name*
alias *name wordlist*

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

alloc

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

bind
bind *key*
bind *function-name key*
bind emacs
bind vi
bind default

An older version of **bindkey** (see below.) **bind** lists key bindings for all keys, **bind** *key* lists the function bound to *key*, **bind** *function-name key* binds the key sequence *key* to the editor function *function-name.* **bind vi, bind emacs,** and **bind default** binds the key sequences to
vi, emacs, or the default sequences, respectively..

key may be a direct character, caret-<letter> for a control character, M-<letter> for an escaped character, or F-<string> for a function key. For a function key, the function key prefix must be bound to the function *sequence-lead-in* and the string specified to **bind** command must not include this prefix.

bindkey

bindkey **–a** *in-string*
bindkey **–s** *in-string out-string*
bindkey **–v**
bindkey **–e**
bindkey **–d**
bindkey **–l**
bindkey **–r** *in-string*
bindkey *in-string*
bindkey *in-string command*

Redefines what a key may do. With out any arguments, lists all the existing bindings, if only an *in-string* is given, the bindings for *in-string* are listed. Otherwise, *in-string* is bound to *command*, or *out-string*, if **–s** is given as an option. *out-string*s are treated as input to csh when *in-string* is entered. This may be used recursively to a depth of 10 levels.

bindkey –l lists the commands, with descriptions, that can be assigned to key sequences. **bindkey –r** removes *in-string* from the binding tables.

bindkey –v, bindkey –e, and **bindkey -d** bind the default editor mode to **vi**, **emacs**, or compiled default (in this implementation, **emacs**) keymaps. There are two key maps in each mode, the normal and the alternative. **vi** mode uses the alternative for **vi** command mode. For multi-character input, the basic key maps contains a sequence-lead-in for the first character in the input.

In strings, control characters may be written as caret-<letter>, and backslash ("\") is used to escape a character as follows:

\a – bell character;

\n – new line (line feed);

\b – back space;

\t – horizontal tab;

\v – vertical tab;

\f – form feed;

\r – carriage return;

\e – escape; and

*****nnn* – octal character *nnn*.

In all other cases \ escapes the following character. This is needed for escaping \ and . The delete character is written as **^?** (caret-question mark.)

The terminal arrow keys, as defined by the termcap entry are always bound as follows: up arrow to up-history; down arrow to down-history; right arrow to forward-char; and left arrow to backward-char. In addition, the vt100 sequence for arrow keys are always bound.

bg
bg *%job ...*

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

break

Resumes execution 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 *directory*
chdir
chdir *directory*
Change the shell's working directory to directory *directory*. If no argument is given, then change to the home directory of the user. If *directory* 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 *directory*. Finally, if all else fails, but *directory* 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 remaining 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.

echotc *cap*
echotc *cap params*
Provides the user with a method of accessing the terminal capabilities database (*termcap*(5)) from the command line. *cap* is a *termcap*(5) capability, and *params* are any parameters needed by that capability, such as row and column for the cursor motion capability. For example, to clear the screen, and place the cursor at the top of the screen, the following command would be entered:

% **echotc home**

To place the cursor at the column 3, row 10, the following command would be entered:

% **echotc cm 3 10**

To recover the sequences to start and end writing to the status line (assuming that your terminal has one), the following commands could be used:

% **set sl="`echotc ts`"**
% **set el="`echotc fs`"**

In addition, **echotc** understands the arguments **baud, lines, cols, meta,** and **tabs. baud** prints the speed of the terminal, **lines** prints the number of lines available on the terminal, **cols** prints the number of columns available on the terminal, and **meta** and **tabs** return a **yes** or **no** indicating the terminals support for either of those capabilities.
Note: Termcap strings may contain wildcard characters, and echoing them will not work correctly. The suggested method of setting shell variables to terminal capability strings is

using double quotes.

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. See *tset*(1) for an example of using *eval*.

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 this command is read from the terminal, the loop is read up once prompting with **?** before any statements in the loop are executed. If you make a mistake entering in a loop at the terminal, you can rub it out.

glob *wordlist*
glob is like **echo,** but no \ escapes are recognized and words are delimited by null characters in the output. This is useful for programs that wish to use the shell to filename expand a list of words..

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

hashstat
Prints a statistics line indicating how effective the internal hash table has been at locating commands and avoiding **execs.** An **exec** is attempted for each component of the *path,* where the hash function indicates a possible hit, and in each component that does not begin with a /.

history
history *n*
history **-h** *n*

history **–r** *n*
history **–t** *n*

Displays the history event list; if *n* is given, only the *n* most recent events are printed. 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**. The **–r** option reverses the order of printout to be most recent first. The **–t** option suppresses the printing of the time stamp in the history list.

if *(expr) command*

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

if *(expr)* **then**
...
else if *(expr2)* **then**
...
else
...
endif

If the specified *expr* is true, then the commands to the first **else** are executed; **else if** *expr2* is true then the commands to the second else are executed, etc. Any number of 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, entering **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 is sent a CONT (continue) signal as well.

The terminate signal kills processes that do not catch the signal; **kill –9 ...** is a sure kill, as the KILL (9) signal cannot be caught. The killed process must belong to the current user unless the user is the super user.

limit
limit *resource*
limit *resource maximum-use*

Limits the consumption by the current process and each process it creates to not individually exceed *maximum-use* on the specified *resource*. If no *maximum-use* is given, then the current limit is printed; if no *resource* is given, then all limitations are given.

Resources that are controllable currently include *cputime* (the maximum number of cpu-seconds to be used by each process), *filesize* (the largest single file that can be created),

datasize (the maximum growth of the data+stack region via *sbrk*(2) beyond the end of the program text), *stacksize* (the maximum size of the automatically-extended stack region), *coredumpsize* (the size of the largest core dump that will be created), *memoryuse* (the maximum size a processs resident set. size may grow to), and *concurrency* (the maximum number of threads a process is allowed to have running in parallel).

The *maximum-use* may be given as a floating-point or integer number followed by a scale factor. For all limits other than *cputime* and *concurrency*, the default scale is **k** or **kilobytes** (1024 bytes); a scale factor of **m** or **megabytes** may also be used. For *cputime*, the default scaling is **seconds**, while **m** for minutes or **h** for hours, or a time of the form *mm:ss* giving minutes and seconds may be used. There is no need for a scale factor for *concurrency*, since the values will be integers between one and the number of processors on the system, inclusively.

For both *resource* names and scale factors, unambiguous prefixes of the names suffice.

log

Informs the user of all users and terminals affected by the *watch* variable, whether they have been announced by *watch* or not.

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.

ls-F

Produces an **ls -aF** style listing if *showdots* is set, otherwise, it produces an **ls -F** style listing. If other flags are passed to **ls-F** the normal ls is executed. **ls-F** appends the following characters depending upon file type:

@	symbolic link to a non-directory.
>	symbolic link to a directory.
&	symbolic link with no target.
=	AF_UNIX domain socket.
|	named pipe.
%	character special device.
#	block special device.
/	directory.
*	executable.

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 superuser may specify negative niceness by using **nice**-number .... command is always executed in a subshell; *command* must be a simple command, not a pipeline, an alias, a command list, or a parenthesized command.

nohup
nohup *command*

The first form can be used in shell scripts to cause hang ups to be ignored for the remainder of the script. The second form causes the specified command to be run with hang ups 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 that are to terminate shell scripts or to return to the terminal command input level. The second form **onintr** – causes all interrupts to be ignored. The final form causes the shell to execute a **goto label** when an interrupt is received or a child process terminates because it was interrupted.

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

popd
popd *+n*

Pops the directory stack, returning to the new top directory. With an 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 *directory*
pushd +n

With no arguments, **pushd** exchanges the top two elements of the directory stack. Given a **name** argument, **pushd** pushes the current working directory (as in *cwd*) onto the directory stack, then changes to the new directory (ala *cd*). With a numeric argument, 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.

sched
sched *[+]hh:mm command*
sched - *n*

arranges for the shell to automatically execute commands for the user at a specified time. The first form displays the commands currently scheduled, while the second schedules *command* to run at *hh:mm* or in *+hh:mm* This allows both absolute and relative scheduling, and absolute scheduling supports am/pm specification for use with 12 hour clocks.

The third form of sched allows the removal of scheduled events, where *n* is the event to remove.

Scheduled commands execute prior to the printing of the first prompt immediately after the time when the command is to be run. Scheduled commands will not interrupt the execution of already running commands, but schedule events that become due while waiting at for

input at the shell prompt will be executed immediately.

Examples:

```
% sched +0:30 /usr/lib/uucp/uucico -r1 -sother

% sched 11:00 echo It's eleven o'clock

% sched 5pm set prompt='[%h] It's after 5; Go Home: > '

% sched
    1  Sat Jul 20 11:00  echo "It's eleven o'clock"
    2  Sat Jul 20 17:00  set prompt='[%h] It's after 5; Go Home: > '

% sched -2

% sched
    1  Sat Jul 20 11:00  echo "It's eleven o'clock"
```

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

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

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

setenv *name value*

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 variables *user*, *term*, and *path*; there is no need to use *setenv* for these.

settc *cap value*

Used for testing *termcap*(5) entries, **settc** tells the shell that the terminal capability *cap* (as defined in *termcap*(5)) has the value *value*. No checking for sanity is performed, so beware of proper use. **settc** is a companion to **telltc**, below.

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 *file*
source **–h** *file*

The shell reads commands from *file*. **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. Note that since **source** is a *csh* builtin function, ^Z and ^C

will not work in it. Attempting to stop or suspend the job will result in an attempt to stop or suspend the shell.

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

telltc
Returns the *termcap*(5) capabilities the shell believes your terminal supports. Used for testing new *termcap*(5) entries. See also **settc**.

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

umask
umask *value*
The file creation mask is displayed (first form) or set to the specified value (second form). The mask is given in octal. Common values for the mask are 002, giving all access to the group and read and execute access to others, or 022, giving yourself all access, but 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 if nothing is *unaliased.*

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

unlimit *resource*
unlimit
Removes the limitation on *resource.* If no *resource* is specified, then all *resource* limitations are removed.

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 names match the specified pattern from the environment. See 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.

whence *name* ...
whence **–u** *name* ...
whence **–w** *name* ...
Searches aliases, builtins, and finally the user's *path* for the program file that would be executed if *name* were to be typed as a command. The output can be in one of three forms: the full path of the command that would be executed, an indication of the wordlist that *name* is aliased to, or notification that the command in question is built in to the shell. If no output is given, the command was not found anywhere in the user's *path*, as a C shell builtin, or as an alias. *whence* may be invoked with multiple names, and each one will be processed in order. If the second form is used, the output will be in the same style as *which*(1). The first form is perhaps more complete since it reports on C shell builtin commands, but both forms run much faster than the *which*(1) utility.

Examples:

```
% whence which
/usr/ucb/which
% alias h history
% whence h
alias/h 'history'
% whence history
builtin/history
% whence -u h
alias/h 'history'
% whence -w h
history: shell built-in command
```

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

%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 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, however, are mandatory in separating components of *expr* that would otherwise be single words.

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

Predefined and environment variables

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

csh copies the environment variable USER into the variable *user,* TERM into *term,* SHLVL into *shlvl,* and HOME into *home,* and copies these back into the environment whenever the normal shell variables are reset. The environment variable PATH is likewise handled. It is not necessary to worry about its setting other than in the file *.cshrc,* as inferior *csh* processes import the definition of *path* from the environment and re-export it if you change it.

Other environment variables of interest to *csh* are HPATH, which contains the path to command documentation, and NOREBIND, which inhibits the rebinding of printable characters to self-insert-command.

addsuffix add a / for directories, and a space for normal files when complete matches a name exactly. If unset don't add anything extra.

ampm show all times in 12 hour, AM/PM format.

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.

autoexpand invoke the expand-history function automatically on completion.

autolist list possibilities on an ambiguous completion. With autolist=beepnone, listing only beeps when there is no match, with autolist=beepnever, it never beeps.

autologout Set to the number of minutes an interactive shell can be idle before the shell automatically terminates itself. The default value for *autologout* is 0; *autologout* disabled. For the shell, idle time is defined to be the time between when the *prompt* is printed and when a command is entered. The *autologout* feature is disabled by setting its value to 0.

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

chase_symlinks always resolve symbolic links to real names on cd, etc.

correct automatically try to correct the spelling of commands. Must be set to either correct=cmd, only command name will be corrected, or correct=all, the whole line will be corrected.

cwd the full pathname of the current directory.

dextract extract and reinsert the directory on the top of the directory stack instead of

rotating the requested directory to the top of the stack.

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 done selectively.

edit use the input editor, set by default.

fignore list of file name suffixes (e.g. .o, ~) to ignore during complete.

gid the current real group id.

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.

histlit If set, history lines in the editor will be shown with its literal value (that is the line as it was input) instead of the shells lexical version. The current history line can be toggled between literal and lexical with the toggle-literal-history function. History lines saved at shell exit are also saved as this variable indicates.

history Can be given a numeric value to control the size of the history list. Any command that has been referenced in this many events is not 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.

ignore_symlinks
don't resolve symbolic links to real names on cd, etc. Set by default.

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

listjobs list all jobs when suspending. **set listjobs=long** produces long format.

listmax maximum number of items to list without asking first.

mail The files where the shell checks for mail. This is done after each command completion that results 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*.

nobeep do not beep on non-unique expansion or excess edit.

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

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

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

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 pathnames will execute. The usual search path is */usr/convex*, */usr/ucb*, */usr/bin*, */bin*, and ., but this may vary from system to system. *path* is imported from PATH upon startup, and *path* is exported back the environment variable PATH any time that *path* is changed.

A shell which is given neither the **–c** nor the **–t** option normally hashes 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* builtin, or the commands may not be found.

printexitvalue if an interactive program exits non-zero, print the exit value.

prompt The string that is printed before each command is read from an interactive terminal input. Default is **%**, or **#** for the superuser.

The prompt variable supports a number of formatting characters, allowing the user to easily put information such as the current working directory, the current time of day, etc, into the prompt. The available formatting characters are:

%d
%/ the current working directory.

%~ the current working directory. If it starts with $*HOME*, that portion is replaced with a "~".

%c
%.
%c*n*
%.*n* The trailing component of the current working directory. If followed by a digit *n*, the *n* trailing components of the current working directory are printed. If the current working directory starts with $*HOME*, that portion is replaced with a "~".

%C
%C*n* The trailing component of the current working directory, with out "~" substitution. If a digit *n* follows, the *n* trailing components will be printed

%h
%!
! The current history event number.

%M The name of the host, as returned by /bin/hostname.

%m The name of the host, up to the first dot in the name.

%S Start standout mode, if the terminal supports it.

%s Stop standout mode.

%B Start boldface mode, if the terminal supports it.

%b Stop boldface mode.

%U Start underline mode, if the terminal supports it.

%u Stop underline mode.

%t

%@ The current time of day, in 12 hour, am/pm format.

%T The current time of day, in 24 hour format, as modified by the *ampm* shell variable, above.

\c Inserts the escape sequence *n* into the prompt string. **\a** inserts the bell character, **\n** inserts a newline, **\b** inserts a backspace, **\t** inserts a tab, **\v** inserts a vertical tab, **\f** inserts a formfeed, **\r** inserts a carriage return, **\e** inserts an escape, and ***nnn*** inserts the character described bye the octal code *nnn*. In all other cases, **** escapes the following character.

^*c* Insert control character *c* into the prompt string.

%% a single % character.

%n The contents fo *$usr*.

%w The date in <Month> <Day> (MMM DD) format.

%W the date in mm/dd/yy format.

%D the date in yy-mm-dd format.

%l The terminal line currently logged into.

%L clear from prompt to end of display or end of line.

%# A # if running as a root shell, a % if not.

%{...%} Include string a a literal escape sequence. Note: The enclosed escape sequence should not move the cursor position. It should only be used to change terminal attributes. It may not be the last character of a prompt string.

%? The return of the last command executed.

prompt_ding controls the printing of the string **DING!** at the top of the hour when the **%T**, **%@**, or **%t** prompt formatting string is used.

prompt2 the string to prompt for *while* and *for* loops with.

prompt3 the string to prompt with when automatic spelling correction has corrected a command line.

pushdtohome make pushd with no args do a "pushd ~" (like cd does).

pushdsilent do not print the dir stack on every pushd and popd.

recexact recognize exact matches even if they are ambiguous.

recognize_only_executables
list choices of commands only displays files in the path that are executable. Setting this may cause command choices to take longer to display.

rmstar prompt the user before execution of 'rm *'.

savedirs if set, saves the current directory stack for the login shell.

savehist 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 that 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. Values of *savehist* that are too large 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 that have execute bits set, but that are not executable by the system. (See the

description of **Non-builtin Command Execution** below.) *Shell* is initialized to the system-dependent home of the shell.

savehist number of history items to save between login sessions.

shlvl Integer value indicating the number of nested shells.

showdots show hidden files in list and complete operations.

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

tcsh Contains the current version of csh as R.VV.PP. The *R* indicates the major release number, the *VV* the current version and the *PP* the patchlevel.

term the terminal type; see above.

time The first word of this array variable is numeric and controls automatic timing of commands. If set, then any command that takes more than this many CPU seconds causes a line to be printed that gives a time and resource usage summary (most of the data is from *getrusage*(2)). The second word of this variable, which takes the form of a *printf*-like format string, can be used to customize the output of the *time* builtin. The following conversion specifications are recognized and have the given meanings:

- **%U** Amount of time spent executing in user mode (in seconds).
- **%S** Amount of time spent in the system executing on behalf of the process (in seconds).
- **%E** Real (elapsed) time in minutes and seconds.
- **%P** Percentage of CPU utilization (which is the ratio of user plus system times to real time, scaled by the number of processors on the system).
- **%C** CPU parallelization factor or concurrency level (somewhere between one and the number of processors on the system). This value is designed to be used when timing a particular process and so will display *N/A* when *time* is issued as a command with no arguments.
- **%W** Number of process swaps out of main memory.
- **%X** Amount of memory shared among other processes (in kilobytes).
- **%D** Combined size of unshared data and stack segments (in kilobytes).
- **%K** Total size of shared memory, unshared data, and unshared stack sizes.
- **%M** Maximum resident set size utilized (in kilobytes).
- **%F** Number of page faults serviced which required I/O activity.
- **%R** Number of page faults serviced without I/O activity; here, I/O activity is avoided by reclaiming a page from the list of pages awaiting reallocation.
- **%I** Number of times the file system had to perform block input.
- **%O** Number of times the file system had to perform block output.
- **%r** Number of socket messages received
- **%s** Number of socket messages sent.

%k Number of signals received.

%w Number of voluntary context switches (waits).

%c Number of involuntary context switches.

The default format string for *time* output is

%Uu %Ss %E %P %X+%Dk %I+%Oio %Fpf+%Ww,

which will print user, system, and real times; a utilization percentage; the amount of shared memory; the combined size of unshared data and stack segments; the number of times the file system had to perform block input and output; the number of page faults; and the number of process swaps.

Examples:

```
% set time = 0
% date
Sat Jun 21 16:30:00 CDT 1986
0.00u 0.01s 0:00 60% 0+0k 1+0io 10pf+0w
% set time = (0 "user=%U sys=%S real=%E; %C|")
% date
Sat Jun 21 16:30:07 CDT 1986
user=0.00 system=0.01 real=0:00; 1.3|
```

tperiod the delay period (in minutes) between execution of the contents of the **periodic** alias.

tty The name of the tty, or empty if not attached to one.

uid the current real user ID.

version the version ID stamp for this **csh.** It contains, the origin of this version of csh, the date this version was released and a string containing a comma separated list of the compile time options enabled:

8b csh was compiled to be eight bit clean or not.

nls csh uses the Native Language System, and should be the default for systems that support it.

el csh executes /etc/login and /etc/logout if the shell is a login shell.

dl csh places the current directory (".") last on the path for security.

al autologout is available.

dir directory stack save and restore is available.

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

visiblebell use the visible bell (screen flash) rather than audible bell.

watch The list of user id/terminal pairs to watch for login or logout events. The delay between checks is 10 minutes. This can be changed by listing the delay time as the first element of the list, much like the **mail** variable. The string "any" is a wildcard for both users and terminals, allowing watching for anyone logging into a particular terimal, or a particular user logging into any terminal. The default for watch is unset. When **watch** is first set, it displays all users currently on the system.

Examples:

Watch for the user "brad" to log in or out on "tty1f".

% **set watch=(brad tty1f)**

Watch for any user to log in or out on "tty3c".

% **set watch=(any tty3c)**

The most general case, watch for anyone logging in or out anywhere.

% **set watch=(any any)**

Check who has logged in or out every five minutes, instead of the default ten.

% **set watch=(5 any any)**

who format string for the print outs generated when *watch* is set. Valid formatting characters are:

%n	the name of the user that logged in or out.
%a	The action that the user took, "logged on", "logged out", or "replaced <olduser> on".
%l	The terminal the user logged in to.
%S	Start standout mode, if the terminal supports it.
%s	Stop standout mode.
%B	Start boldface mode, if the terminal supports it.
%b	Stop boldface mode.
%U	Start underline mode, if the terminal supports it.
%u	Stop underline mode.
%M	The full hostname of the remote host ("local" if non-remote)
%m	The short hostname, up to the first dot, unless only the hostname is an IP address or an X Windows display.
%t	
%@	The time of the login/logout event in 12 hour, am/pm format.
%T	The time of the event in 24 hour format, as modified by the *ampm* shell variable, above.
%w	The date in <Month> <day> (MMM DD) format.
%W	The date in mm/dd/yy format.
%D	The date in yy-mm-dd format.

wordchars list of nonalphanumeric characters considered part of a word for the purpose of the forward-word, backward-word etc functions -- defaults to "*?_-.[]~=".

Predefined aliases

The following aliases are predefined to the shell, and when set, cause an action to take place automatically.

cwdcmd If set, this alias is executed after every change of working directory. This is useful for changing status lines to contain the current working directory, or any of many other (probably as yet unthought of) things.

periodic If set, alias is executed every *tperiod* minutes. This provides a convenient means for checking on common but infrequent changes, such as new messages. The shell variable *tperiod* must also be set. When *tperiod* is set to zero, **periodic** acts like **precmd**, below.

precmd If set, this alias is executed prior to the printing of each prompt.

Non-builtin command execution

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

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

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

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

Argument list processing

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

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

–e The shell exits if any invoked command terminates abnormally or yields a nonzero exit status.

–f The shell starts faster because it neither searches for nor executes commands from the file *.cshrc* in the invoker's home directory.

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

–n Commands are parsed, but not executed. This 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.**

After processing of flag arguments, if 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 **&**, **bg**, or **%... &** commands) are immune to signals generated from the keyboard, including hang ups. Other signals have the values that the shell inherited from its parent. The shells handling of 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 shells parent. In no case are interrupts allowed when a login shell is reading the file *.logout.*

FILES

~/.cshrc	Read at beginning of execution by each shell.
~/.login	Read by login shell after */etc/login* 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/login	Read by login shell after *.cshrc* at login.
/etc/logout	Read by login shell after *.logout* at logout.
/etc/passwd	Source of home directories for **~name**.

LIMITATIONS

Words can be no longer than 1024 characters. The system limits argument lists to 10240 characters. The number of arguments to a command, which involves filename expansion, is limited to 1/6 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), killpg(2), pipe(2), sigvec(2), umask(2), getrlimit(2), wait(2), tty(4), a.out(5), environ(7)

"An Introduction to the C Shell" in the *ConvexOS Tutorial Papers* "ConvexOS Primer"

BUGS

When a command is restarted from a stop, the shell prints the directory it started in, if this is different from the current directory. This can be misleading (i.e., wrong) because the job may have changed directories internally.

Shell builtin functions are not stoppable or restartable. Command sequences of the form **a ; b ; c** are also not handled gracefully when stopping is attempted. If you suspend **b**, the shell immediately executes **c**. This is especially noticeable if this expansion results from an *alias*. It suffices to place the sequence of commands in **( )** 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. In a virtual terminal interface, much more interesting 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 builtin 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.

The **-x** test for executability can be fooled (see the **Expressions** section for more information).

NOTES

This version of the csh runs in raw mode, but does tty sanity checking to maintain a reasonable tty state at all times.

Even though this version of csh runs in raw mode, type ahead is allowed and accepted.

Although called csh on ConvexOS, this shell is really **tcsh**, based upon version 5.20.02.

Many thanks to Christos Zoulas of Cornell University, the current **tcsh** maintainer, and the many people who have worked on **tcsh** over the years.

NAME

ctags – create a tags file

SYNOPSIS

ctags [**–BFatuwvx**] name ...

DESCRIPTION

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

If the **–x** flag is given, *ctags* produces a list of object names, the line number and filename on which each is defined, as well as the text of that line and prints this on the standard output. This is a simple index which can be printed out as an offline readable function index.

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

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

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

Other options are:

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

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

–a append to *tags* file.

–t create *tags* for *typedefs*.

–w suppressing warning diagnostics.

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

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

FILES

tags	output *tags* file

SEE ALSO

ex(1), vi(1)

BUGS

Recognition of functions and subroutines for FORTRAN is done in a very elementary way.

The method of deciding whether to look for C and FORTRAN functions is not sophisticated.

Does not know about #ifdefs.

Relies on the input being well formed to detect *typedefs*. Use of **-tx** shows only the last line of *typedefs*.

The **–v** option is of little use. *Vgrind*(1) is an unsupported product.

NAME

date – print and set the date

SYNOPSIS

date [-u] [-z zone] [yymmddhhmm [.ss]]

DESCRIPTION

If no arguments are given, the current date and time are printed. If a date is specified, the current date is set. The *-u* flag is used to display the date in GMT (universal) time. This flag may also be used to set GMT time. *yy* is the last two digits of the year; 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; *.ss* is optional and is the seconds. For example:

date 10080045

sets the date to Oct 8, 12:45 AM. The year, month and day may be omitted, the current values being the defaults. The system operates in GMT. *Date* takes care of the conversion to and from local standard and daylight time.

The *-z* flag is used to set the time zone where *zone* is a string which identifies the local time zone. The string can be in one of two forms. The simplest form of the string is a mnemonic for the time zone as follows:

Zone Name	Time Zone
ast | adt	US: Atlantic
est | edt	US: Eastern
cst | cdt	US: Central
mst | mdt	US: Mountain
pst | pdt	US: Pacific
eet | eetdst	Eastern European
met | metdst	Middle European
wet | wetdst	Western European
aest | aedt	Australia: Eastern
acst | acdt	Australia: Central
awst | awdt	Australia: Western

The more general form of the string is

[+|-]hh:mm[,dst_rule_name]

where *hh:mm* is the number of hours and minutes that the local time zone is east *(-)* or west *(+)* of GMT. The optional *dst_rule_name* specifies the daylight savings time rule to be used. If *dst_rule_name* is not specified, then it is assumed that daylight savings time does not apply to the local time zone. The following *dst_rule_name* mnemonics may be used:

Dst_rule_name	DST Rule
us	US
eet	Eastern European
met	Middle European
wet	Western European
aus	Australia

Note that the time zone must be set each time that ConvexOS is booted. The time zone mnemonic displayed by **date** is determined by the daylight savings time rules used in **ctime(3)**. Hence, specifying the time zone to be *cdt* in January will result in EST being displayed by **date** since daylight savings time is not in effect in January for the central time zone in the United States.

The time zone mnemonics displayed by **date** can be modified by specifying the desired mnemonics in the environmental variable TZNAME. For example,

```
TZNAME="STD,DST"; export TZNAME
```

will result in STD and DST being displayed as the standard time and daylight savings time mnemonics, respectively.

FILES

/usr/adm/wtmp to record time-setting

SEE ALSO

utmp(5), ctime(3), tzset(3)

DIAGNOSTICS

'Failed to set date: Not owner' or 'Failed to set time zone: Not owner' if you try to change the date but are not the superuser.

NAME

dbadd, dbcreate, dblist, dbprint – EMACS database manipulation

SYNOPSIS

dbadd dbname key
dbcreate dbname
dblist dbname [**-l**] [**-p** newdbname]
dbprint dbname key

DESCRIPTION

All of these commands deal with databases used by the UNIX EMACS database manipulation facilities. A UNIX EMACS database is simply a set of (key,content) pairs, as in *dbm*(3), except that the content portion can be a very long string.

Dbadd adds the text from the standard input to the named database using the given key. **Dbcreate** creates the named database, making it empty. **Dbprint** prints the contents of the entry from the database with the given key.

Dblist with no arguments simply lists the keys of all the items in the database. With the **-l** option it prints some internal information from the database which is of no interest to anyone but the implementor. The **-p** option causes the key and content of every entry to be listed as a shell command file which, when executed, will repeatedly invoke **dbadd** to rebuild the database. This form of **dblist** is handy when you want a readable ascii file representation of a data base for shipping around or editing. Databases should be recreated periodically to clean them out.

FILES

dbname.dir, *dbname*.pag, and *dbname*.dat: the three component subfiles of a database.

SEE ALSO

UNIX EMACS Manual
emacs(1), dbm (3x)

AUTHOR

James Gosling

NAME

dc – desk calculator

SYNOPSIS

dc [file]

DESCRIPTION

Dc is an arbitrary precision arithmetic package. Ordinarily it operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained. The overall structure of *dc* is a stacking (reverse Polish) calculator. If an argument is given, input is taken from that file until its end, then from the standard input. The following constructions are recognized:

number
: The value of the number is pushed on the stack. A number is an unbroken string of the digits 0-9. It may be preceded by an underscore _ to input a negative number. Numbers may contain decimal points.

+ - / * % ^
: The top two values on the stack are added (+), subtracted (-), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. Any fractional part of an exponent is ignored.

s*x*
: The top of the stack is popped and stored into a register named *x*, where *x* may be any character. If the **s** is capitalized, *x* is treated as a stack and the value is pushed on it.

l*x*
: The value in register *x* is pushed on the stack. The register *x* is not altered. All registers start with zero value. If the **l** is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

d
: The top value on the stack is duplicated.

p
: The top value on the stack is printed. The top value remains unchanged. **P** interprets the top of the stack as an ascii string, removes it, and prints it.

f
: All values on the stack and in registers 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 executed 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 ConvexOS 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.

An example which prints the first ten values of n! is

```
[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.
"DC--An Interactive Desk Calculator" in the *ConvexOS Tutorial Papers*

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

SYNOPSIS

dd [option=value] ...

DESCRIPTION

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

option	*values*
if=	input file name; standard input is default
of=	output file name; standard output is default
ibs=*n*	input block size *n* bytes (default 512)
obs=*n*	output block size (default 512)
bs=*n*	set both input and output block size, superseding *ibs* and *obs;* also, if no conversion is specified, it is particularly efficient since no copy need be done
cbs=*n*	conversion buffer size
skip=*n*	skip *n* input records before starting copy
files=*n*	copy *n* input files before terminating (makes sense only where input is a magtape or similar device).
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
block	convert variable length records to fixed length
unblock	convert fixed length records to variable length
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.

The conversion options *ascii, unblock, ebcdic, ibm,* and *block* , are, for the most part, mutually exclusive, in that the last one specified is the one which takes effect. The only exception to this is that the *ebcdic* conversion implies the *block* conversion -- therefore, *conv=block,ebcdic* is redundant. However, if *conv=ebcdic,block* is given, the *block* will override the *ebcdic* option, and the data will undergo only *block* conversion.

If both the *lcase* and *ucase* conversions are specified, only the *lcase* conversion will be applied, independent of the order they are specfied in.

The *swab, noerror,* and *sync,* conversions are cumulative and do not interfere with each other.

Cbs is used only if *ascii, unblock, ebcdic, ibm,* or *block* conversion is specified. In the first two cases, *cbs* characters are placed into the conversion buffer, any specified character mapping is done, trailing blanks trimmed and new-line added before sending the line to the output. In the latter three cases, characters are read into the conversion buffer, 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.

For example, to read an EBCDIC tape blocked ten 80-byte EBCDIC card images per record into the ASCII file *x*:

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

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

NOTES

dd is capable of reading and writing files of larger that two gigabytes of data in size.

SEE ALSO

cp(1), tr(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.

One must specify "conv=noerror,sync" when copying raw disks with bad sectors to insure *dd* stays synchronized.

NAME

deroff – remove nroff, troff, tbl and eqn constructs

SYNOPSIS

deroff [**–w**] file ...

DESCRIPTION

Deroff reads each file in sequence and removes all *nroff* and *troff* command lines, backslash constructions, macro definitions, *eqn* constructs (between '.EQ' and '.EN' lines or between delimiters), and table descriptions and writes the remainder on the standard output. *Deroff* follows chains of included files ('.so' and '.nx' commands); if a file has already been included, a '.so' is ignored and a '.nx' terminates execution. If no input file is given, *deroff* reads from the standard input file.

If the **–w** flag is given, the output is a word list, one 'word' (string of letters, digits, and apostrophes, beginning with a letter; apostrophes are removed) per line, and all other characters ignored. Otherwise, the output follows the original, with the deletions mentioned above.

SEE ALSO

nroff(1), troff(1)(optional product), eqn(1), tbl(1)

BUGS

Deroff is not a complete *troff* interpreter, so it can be confused by subtle constructs. Most errors result in too much rather than too little output.

NAME

df – disk free

SYNOPSIS

df [**–a**] [**–i**] [**–b**] [**–t** *type*] [*filesystem ...*] [*file ...*]

DESCRIPTION

df reports the amount of free disk space available on the specified *filesystem*, e.g. */dev/rda0a*, or on the *filesystem* in which the specified *file*, e.g. "$HOME", is contained. If no *filesystem* is specified, the amount of free space on all of the normally mounted filesystems is reported. The reported numbers are in kilobytes.

Other options are:

–a Report on all filesystems including those which have zero total blocks. (e.g. *automounter*)

–i Report the number of inodes that are used and free and the filesystem block and fragment size. This information is not reported for NFS-mounted filesystems due to possible incompatibilities between filesystem types.

–b Same output as **–i** option (for compatibility).

–t *type* Report on filesystems of a given *type* (for example, **nfs** or **4.2**).

FILES

/etc/mtab list of normally mounted filesystems

SEE ALSO

du(1), mtab(5), quot(8)

NAME

diction,explain – print wordy sentences; thesaurus for diction

SYNOPSIS

diction [**–ml**] [**–mm**] [**–n**] [**–f** pfile] file ...
explain

DESCRIPTION

Diction finds all sentences in a document that contain phrases from a data base of bad or wordy diction. Each phrase is bracketed with [ ]. Because *diction* runs *deroff* before looking at the text, formatting header files should be included as part of the input. The default macro package **–ms** may be overridden with the flag **–mm.** The flag **–ml** which causes **deroff** to skip lists, should be used if the document contains many lists of non-sentences. The user may supply her/his own pattern file to be used in addition to the default file with **–f** *pfile*. If the flag **–n** is also supplied the default file will be suppressed.

Explain is an interactive thesaurus for the phrases found by diction.

SEE ALSO

deroff(1)

BUGS

Use of non-standard formatting macros may cause incorrect sentence breaks. In particular, *diction* doesn't understand **–me.**

NAME

diff, diffh – differential file and directory comparator

SYNOPSIS

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

DESCRIPTION

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

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

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

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

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

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

There are several options for output format; the default output format contains lines of these forms:

n1 a *n3,n4*
n1,n2 d *n3*
n1,n2 c *n3,n4*

These lines resemble *ed* commands to convert *file1* into *file2*. The numbers after the letters pertain to *file2*. In fact, by exchanging 'a' for 'd' and reading backward, one may ascertain equally how to convert *file2* into *file1*. As in *ed*, identical pairs where *n1* = *n2* or *n3* = *n4* are abbreviated as a single number.

Following each of these lines come all the lines that are affected in the first file flagged by '<', then all the lines that are affected in the second file flagged by '>'.

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

–e Produces a script of *a, c,* and *d* commands for the editor *ed*, which will recreate *file2* from *file1*. In connection with **–e**, the following shell program may help maintain multiple versions of a file. Only an ancestral file ($1) and a chain of version-to-version *ed* scripts ($2,$3,...) made by *diff* need be on hand. A 'latest version' appears on the standard output:

```
(shift; cat $*; echo '1,$p') | ed - $1
```

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

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

–n produces a script similar to that of **–e**, but in the opposite order and with a count of changed lines on each insert or delete command. This is the form used by *rcsdiff*(1).

–c Produces a *diff* with lines of context. The default is to present three lines of context and may be changed, e.g to 10, by **–c10**. With **–c** the output format is modified slightly: the output, beginning with identification of the files involved and their creation dates, and then each change is separated by a line with a dozen *'s. The lines

removed from *file1* are marked with '–'; those added to *file2* are marked '+'. Lines which are changed from one file to the other are marked in both files with '!'.

–h Does a fast, half-hearted job. It works only when changed stretches are short and well separated, but does work on files of unlimited length.

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

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

–w is similar to **–b** but causes whitespace (blanks and tabs) to be totally ignored. E.g., "if (a == b)" will compare equal to "if(a==b)".

–i ignores the case of letters. E.g., "A" will compare equal to "a".

–t will expand tabs in output lines. Normal or **–c** output adds character(s) to the front of each line which may screw up the indentation of the original source lines and make the output listing difficult to interpret. This option will preserve the original source's indentation.

FILES

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

SEE ALSO

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

DIAGNOSTICS

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

BUGS

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

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

NAME

diff3 – 3-way differential file comparison

SYNOPSIS

diff3 [**–exEX3**] file1 file2 file3

DESCRIPTION

Diff3 compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

=====	all three files differ
=====1	*file1* is different
=====2	*file2* is different
=====3	*file3* is different

The type of change suffered in converting a given range of a given file to some other is indicated in one of these ways:

f : *n1* **a**	Text is to be appended after line number *n1* in file *f*, where *f* = 1, 2, or 3.
f : *n1* , *n2* **c**	Text is to be changed in the range line *n1* to line *n2*. If *n1* = *n2*, the range may be abbreviated to *n1*.

The original contents of the range follows immediately after a **c** indication. When the contents of two files are identical, the contents of the lower-numbered file is suppressed.

Under the **–e** option, *diff3* publishes a script for the editor *ed* that will incorporate into *file1* all changes between *file2* and *file3*, *i.e.*, the changes that normally would be flagged ===== and =====3. Option **–x** (**–3**) produces a script to incorporate only changes flagged ===== (=====3). The following command will apply the resulting script to 'file1'.

```
(cat script; echo '1,$p') | ed – file1
```

The **–E** and **–X** are similar to **–e** and **–x**, respectively, but treat overlapping changes (i.e., changes that would be flagged with ===== in the normal listing) differently. The overlapping lines from both files will be inserted by the edit script, bracketed by "<<<<<<<" and ">>>>>>>" lines.

For example, suppose lines 7-8 are changed in both file1 and file2. Applying the edit script generated by the command

"diff3 -E file1 file2 file3"

to file1 results in the file:

```
lines 1-6
of file1
<<<<<<< file1
lines 7-8
of file1
=========
lines 7-8
of file3
>>>>>>> file3
rest of file1
```

The **–E** option is used by RCS *merge*(1) to insure that overlapping changes in the merged files are preserved and brought to someone's attention.

FILES

/tmp/d3?????
/usr/lib/diff3

SEE ALSO

diff(1)

BUGS

Text lines that consist of a single '.' will defeat **–e**.

NAME

du – summarize disk usage

SYNOPSIS

du [**–s**] [**–a**] [**–l**] [name ...]

DESCRIPTION

Du gives the number of kilobytes used by all files and, recursively, directories within each specified directory or file *name*. If *name* is missing, **.** is used. The actual number of disk blocks used is counted, which is different than the file size reported by *ls*(1).

The argument **–s** causes only the grand total to be given. The argument **–a** causes an entry to be generated for each file. Absence of either causes an entry to be generated for each directory only. The argument –l inhibits recursive descents of directories which are mounted from remote machines via the Network File System.

If *du* is given an argument *softlink*, the link is not followed but the argument *softlink/* is followed. This is because the two arguments are not functionaly identical. The trailing slash indicates the link should be followed.

A file which has two links to it is only counted once.

SEE ALSO

df(1), diskuse(8), quot(8),
"Accounting" chapter in the *CONVEX System Manager's Guide.*

BUGS

Non-directories given as arguments (not under **–a** option) are not listed.

If there are too many distinct linked files, *du* counts the excess files multiply.

NOTES

NFS is an optional product; for procurement information, contact your CONVEX sales representative.

NAME

echo – echo arguments

SYNOPSIS

echo [**–n**] [arg] ...

DESCRIPTION

Echo writes its arguments separated by blanks and terminated by a newline on the standard output. If the flag **–n** is used, no newline is added to the output.

Echo is useful for producing diagnostics in shell programs and for writing constant data on pipes. To send diagnostics to the standard error file, do 'echo ... 1>&2'.

NAME

ed – text editor

SYNOPSIS — UNITED STATES DISTRIBUTION

ed [–] [**–x**] [name]

SYNOPSIS — INTERNATIONAL DISTRIBUTION

ed [–] [name]

DESCRIPTION

Ed is the standard text editor.

If a *name* 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. If **–x** is present, an *x* command is simulated first to handle an encrypted file. Note: neither the *x* command in *ed* nor the *–x* command line option are supported in the international distribution of ConvexOS. The optional – suppresses the printing of explanatory output and should be used when the standard input is an editor script.

Ed operates on a copy of any file it is editing; changes made in 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*.

Commands to *ed* have a simple and regular structure: zero or more *addresses* followed by a single character *command*, possibly followed by parameters to the command. These addresses specify one or more lines in the buffer. Missing addresses are supplied by default.

In general, only one command may appear on a line. Certain commands allow the addition of text to 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. A regular expression specifies a set of strings of characters. A member of this set of strings is said to be *matched* by the regular expression. In the following specification for regular expressions the word 'character' means any character but newline.

1. Any character except a special character matches itself. Special characters are the regular expression delimiter plus \[. and sometimes ^ * $.
2. A . matches any character.
3. A \ followed by any character except a digit or ( ) matches that character.
4. A nonempty string *s* bracketed [*s*] (or [^ *s*]) matches any character in (or not in) *s*. In *s*, \ has no special meaning, and] may only appear as the first letter. A substring *a–b*, with *a* and *b* in ascending ASCII order, stands for the inclusive range of ASCII characters.
5. A regular expression of form 1-4 followed by * matches a sequence of 0 or more matches of the regular expression.
6. A regular expression, *x*, of form 1-8, bracketed \(*x* \) matches what *x* matches.
7. A \ followed by a digit *n* matches a copy of the string that the bracketed regular expression beginning with the *n*th \(matched.
8. A regular expression of form 1-8, *x*, followed by a regular expression of form 1-7, *y* matches a match for *x* followed by a match for *y*, with the *x* match being as long as possible while still permitting a *y* match.
9. A regular expression of form 1-8 preceded by ^ (or followed by $), is constrained to matches that begin at the left (or end at the right) end of a line.

10. A regular expression of form 1-9 picks out the longest among the leftmost matches in a line.

11. An empty regular expression stands for a copy of the last regular expression encountered.

Regular expressions are used in addresses to specify lines and in one command (see *s* below) to specify a portion of a line which is to be replaced. If it is desired to use one of the regular expression metacharacters as an ordinary character, that character may be preceded by '\'. This also applies to the character bounding the regular expression (often '/') and to '\' itself.

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; however, the exact effect on the current line is discussed under the description of the 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 name *x*, which must be a lowercase letter. Lines are marked with the *k* command described below.

5. A regular expression enclosed in slashes '/' addresses the line found by searching forward from the current line and stopping at the first line containing a string that matches the regular expression. If necessary, the search wraps around to the beginning of the buffer.

6. A regular expression enclosed in queries '?' addresses the line found by searching backward from the current line and stopping at the first line containing a string that matches the regular expression. If necessary, the search wraps around to the end of the buffer.

7. An address followed by a plus sign '+' or a minus sign '-' followed by a decimal number specifies that address plus (resp. 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 (resp. subtracted). As a consequence of this rule and rule 8, the address '-' refers to the line before the current line. Moreover, trailing '+' and '-' characters have a cumulative effect, so '—' refers to the current line less 2.

10. To maintain compatibility with earlier versions of the editor, the character '^' in addresses is equivalent to '-'.

Commands may require zero, one, or two addresses. Commands which require no addresses regard the presence of an address as an error. Commands which accept one or two addresses assume default addresses when insufficient are given. If more addresses are given than such a command requires, the last one or two (depending on what is accepted) are used.

Addresses are separated from each other typically by a comma ','. They may also be separated by a semicolon ';'. In this case the current line '.' is set to the previous address before the next address is interpreted. This feature can be used to determine the starting line for forward and backward searches ('/', '?'). The second address of any two-address sequence must correspond to a line following 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, but are used to show that the given addresses are the default.

As mentioned, it is generally illegal for more than one command to appear on a line. However, most commands may be suffixed by 'p' or by 'l', in which case the current line is either printed or listed, respectively, in the way discussed below. Commands may also be suffixed by 'n', meaning the output of the command is to be line numbered. These suffixes may be combined in any order.

(.)a
<text>
.

The append command reads the given text and appends it after the addressed line. '.' is left on the last line input, if there were any; otherwise at the addressed line. Address '0' is legal for this command; text is placed at the beginning of the buffer.

(.,.)c
<text>
.

The change command deletes the addressed lines, then accepts input text which replaces these lines. '.' is left at the last line input; if there were none, it is left at the line preceding the deleted lines.

(.,.)d

The delete command deletes the addressed lines from the buffer. The line originally after the last line deleted becomes the current line; if the lines deleted were originally at the end, the new last line becomes the current line.

e filename

The edit 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. The number of characters read is typed. 'filename' is remembered for possible use as a default filename in a subsequent *r* or *w* command. If 'filename' is missing, the remembered name is used.

E filename

This command is the same as *e*, except that no diagnostic results when no *w* has been given since the last buffer alteration.

f filename

The filename command prints the currently remembered filename. If 'filename' is given, the currently remembered filename is changed to 'filename'.

(1,$)g/regular expression/command list

In the global command, the first step is to mark every line which matches the given regular expression. Then for every such line, the given command list is executed with '.' initially set to that line. A single command or the first of multiple commands appears on the same line with the global command. All lines of a multi-line list except the last line must be ended with '\'. *A*, *i*, and *c* commands and associated input are permitted; the '.' terminating input mode may be omitted if it would be on the last line of the command list. The commands *g* and *v* are not permitted in the command list.

(.)i
<text>
.

This command inserts the given text before the addressed line. '.' is left at the last line input, or, if there were none, at the line before the addressed line. This command differs from the *a* command only in the placement of the text.

(.,.+1)j

This command joins the addressed lines into a single line; intermediate newlines simply disappear. '.' is left at the resulting line.

(.)k*x*
The mark command marks the addressed line with name *x*, which must be a lowercase letter. The address form "*x*' then addresses this line.

(.,.)l
The list command prints the addressed lines in an unambiguous way: nongraphic characters are printed in two-digit octal, and long lines are folded. The *l* command may be placed on the same line after any non-i/o command.

(.,.)m*a*
The move command repositions the addressed lines after the line addressed by *a*. The last of the moved lines becomes the current line.

(.,.)p
The print command prints the addressed lines. '.' is left at the last line printed. The *p* command may be placed on the same line after any non-i/o command.

(.,.)P
This command is a synonym for *p*.

q The quit command causes *ed* to exit. No automatic write of a file is done.

Q This command is the same as *q*, except that no diagnostic results when no *w* has been given since the last buffer alteration.

($)r filename
The read command reads in the given file after the addressed line. If no filename is given, the remembered filename, if any, is used (see *e* and *f* commands). The filename is remembered if there was no remembered filename already. 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 left at the last line read in from the file.

(.,.)s/regular expression/replacement/ or,
(.,.)s/regular expression/replacement/g
The substitute command searches each addressed line for an occurrence of the specified regular expression. On each line in which a match is found, all matched strings are replaced by the replacement specified, 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 punctuation character may be used instead of '/' to delimit the regular expression and the replacement. '.' is left at the last line substituted.

An ampersand '&' appearing in the replacement is replaced by the string matching the regular expression. The special meaning of '&' in this context may be suppressed by preceding it by '\'. The characters '*n*' where *n* is a digit, are replaced by the text matched by the *n*-th regular subexpression enclosed between '\(' and '\)'. When nested, parenthesized subexpressions are present, *n* is determined by counting occurrences of '\(' starting from the left.

Lines may be split by substituting newline characters into them. The newline in the replacement string must be escaped by preceding it by '\'.

One or two trailing delimiters may be omitted, implying the 'p' suffix. The special form 's' followed by *no* delimiters repeats the most recent substitute command on the addressed lines. The 's' may be followed by the letters **r** (use the most recent regular expression for the left side, instead of the most recent left side of a substitute command), **p** (complement the setting of the *p* suffix from the previous substitution), or **g** (complement the setting of the *g* suffix). These letters may be combined in any order.

(.,.)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 on the last line of the copy.

(.,.)u
The undo command restores the buffer to its state before the most recent buffer modifying command. The current line is also restored. Buffer modifying commands are *a, c, d, g, i, k, m, r, s,* and *v*. For purposes of undo, *g* and *v* are considered to be a single-buffer modifying command. Undo is its own inverse.

When *ed* runs out of memory, this full undo is not possible, and *u* can only undo the effect of the most recent substitute on the current line. This restricted undo also applies to editor scripts when *ed* is invoked with the - option.

(1, $)v/regular expression/command list
This command is the same as the global command *g* except that the command list is executed *g* with '.' initially set to every line *except* those matching the regular expression.

(1, $)w filename
The write command writes the addressed lines onto the given file. If the file does not exist, it is created. The filename is remembered if there was no remembered filename already. If no filename is given, the remembered filename, if any, is used (see *e* and *f* commands). '.' is unchanged. If the command is successful, the number of characters written is printed.

(1, $)W filename
This command is the same as *w,* except that the addressed lines are appended to the file.

x A key string is demanded from the standard input. Later *r, e,* 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. Note: neither the *x* command in *ed* nor the *-x* command line option are supported in the international distribution of ConvexOS.

(.+1)z or,
(.+1)z*n*
This command scrolls through the buffer starting at the addressed line; 22 (or *n,* if given) lines are printed. The last line printed becomes the current line. The value *n* is sticky, in that it becomes the default for future *z* commands.

($)=
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 *sh*(1) to be interpreted as a command. '.' is unchanged.

(.+1,.+1)<newline>
An address alone on a line causes the addressed line to be printed. A blank line alone is equivalent to '.+1p'; it is useful for stepping through text. If two addresses are present with no intervening semicolon, *ed* prints the range of lines. If they are separated by a semicolon, the second line is printed.

If an interrupt signal (ASCII DEL) is sent, *ed* prints '?interrupted' and returns to its command level.

Some size limitations: 512 characters per line, 256 characters per global command list, 64 characters per filename, and 128K characters in the temporary file. The limit on the number of lines depends on the amount of core: each line takes 2 words. The maximum size of a file to be edited is the same as the size of the temporary file. This may vary by as much as 10 to 20k depending on the above considerations.

When reading a file, *ed* discards ASCII NUL characters and all characters after the last newline. It refuses to read files containing non-ASCII characters.

FILES

/tmp/e*
edhup: work is saved here if terminal hangs up

SEE ALSO

"A Tutorial Introduction to the UNIX Text Editor" in the *ConvexOS Tutorial Papers*
ex(1), sed(1)

DIAGNOSTICS

'?' for simple errors.
'?name' for an inaccessible file.
'?File size limit exceeded' for attempting to edit a file that is too large.
'?self-explanatory message' for other errors.

To protect against throwing away valuable work, a *q* or *e* command is considered to be in error, unless a *w* has occurred since the last buffer change. A second *q* or *e* will be obeyed regardless.

BUGS

The *l* command mishandles DEL.
The *undo* command causes marks to be lost on affected lines.
The *x* command, **–x** option, and special treatment of hangups only work on UNIX.

NAME

emacs – GNU project Emacs

SYNOPSIS

emacs [file ...]

DESCRIPTION

GNU Emacs is a new version of *Emacs,* written by the author of the original (PDP-10) *Emacs,* Richard Stallman. Its user functionality encompasses everything other *Emacs* editors do, and it is easily extensible since its editing commands are written in Lisp.

Emacs has an extensive interactive help facility, but the facility assumes that you know how to manipulate *Emacs* windows and buffers. CTRL-h (backspace or CTRL-h) enters the Help facility. Help Tutorial (CTRL-h t) requests an interactive tutorial which can teach beginners the fundamentals of *Emacs* in a few minutes. Help Apropos (CTRL-h a) helps you find a command given its functionality, Help Character (CTRL-h c) describes a given character's effect, and Help Function (CTRL-h f) describes a given Lisp function specified by name.

Emacs's Undo can undo several steps of modification to your buffers, so it is easy to recover from editing mistakes.

GNU Emacs's many special packages handle mail reading (RMail) and sending (Mail), outline editing (Outline), compiling (Compile), running subshells within *Emacs* windows (Shell), running a Lisp read-eval-print loop (Lisp-Interaction-Mode), and automated psychotherapy (Doctor).

There is an extensive reference manual, but users of other Emacses should have little trouble adapting even without a copy. Users new to *Emacs* will be able to use basic features fairly rapidly by studying the tutorial and using the self-documentation features.

Emacs Options

The following options are of general interest:

file Edit *file.*

+number
Go to the line specified by *number* (do not insert a space between the "+" sign and the number).

–d displayname
Create the *Emacs* window on the display specified by *displayname.* This must be the first argument listed in the command line.

–q Do not load an init file.

–u user Load *user's* init file.

–t file Use specified *file* as the terminal instead of using stdin/stdout. This must be the first argument specified in the command line.

The following options are lisp-oriented (these options are processed in the order encountered):

–f function
Execute the lisp function *function.*

–l file Load the lisp code in the file *file.*

The following options are useful when running *Emacs* as a batch editor:

–batch commandfile
Edit in batch mode using the commands found in *commandfile.* The editor will send messages to stdout. This option must be the first in the argument list.

–kill Exit *Emacs* while in batch mode.

Using Emacs with X

Emacs has been tailored to work well with the X window system. To enable this feature, you must define the macro **HAVE_X_WINDOWS** in the file *src/config.h* before compiling *Emacs*. If you run *Emacs* from under X windows, it will create its own X window to display in. You will probably want to start the editor as a background process so that you can continue using your original window. To use the optional X Menu features, define also the macro **HAVE_X_MENU.** This macro is separate from **HAVE_X_WINDOWS** because the Menu facility of X does not work on all the systems that support X. *Emacs* can be started with the following X switches:

–r Display the *Emacs* window in inverse video.

–i Use the "kitchen sink" bitmap icon when iconifying the *Emacs* window.

–font font
Set the *Emacs* window's font to that specified by *font.* You will find the various *X* fonts in the */usr/new/lib/X/font* directory. Note that *Emacs* will only accept fixed width fonts. These include the *6x10.onx, 6x13.onx, 6x13p.onx, 8x13.onx,* and *9x15.onx* fonts. The other fixed width fonts are specified by the *fx* character sequence that comes before the *.onx* extension.

When you specify a font, do not include the *.onx* extension. Be sure to put a space between the *–font* switch and the font specification argument.

–b pixels
Set the *Emacs* window's border width to the number of pixels specified by *pixels.*

–ib pixels
Set the window's internal border width to the number of pixels specified by *pixels.* Defaults to one pixel of padding on each side of the window.

–w =[WIDTH][xHEIGHT][{+-}XOFF[{+-}YOFF]]
Set the *Emacs* window's width, height, and position on the screen. The []'s denote optional arguments, the {}'s surround alternatives. WIDTH and HEIGHT are in number of characters, XOFF and YOFF are in pixels. WIDTH defaults to 80, HEIGHT to 24, XOFF and YOFF to 1. If you don't give XOFF and/or YOFF, then you must use the mouse to create the window. If you give XOFF and/or YOFF, then a WIDTHxHEIGHT window will automatically be creating without intervention. XOFF and YOFF specify deltas from a corner of the screen to the corresponding corner of the window, as follows:

+XOFF+YOFF	upper left to upper left
–XOFF+YOFF	upper right to upper right
+XOFF-YOFF	lower left to lower left
–XOFF-YOFF	lower right to lower right

–fg color
On color displays, sets the color of the text.

–bg color
On color displays, sets the color of the window's background. See the file */usr/lib/rgb.txt* for a list of valid *color* names.

–bd color
On color displays, sets the color of the window's border.See the file */usr/lib/rgb.txt* for a list of valid *color* names.

–cr color
On color displays, sets the color of the window's text cursor. See the file */usr/lib/rgb.txt* for a list of valid *color* names.

–ms color

On color displays, sets the color of the window's mouse cursor. See the file */usr/lib/rgb.txt* for a list of valid *color* names.

–d displayname
Create the *Emacs* window on the display specified by *displayname*. Must be the first option specified in the command line. **–nw** Tells *Emacs* not to use its special interface to X. If you use this switch when invoking *Emacs* from an *xterm* window, display is done in the *xterm* window. This must be the first option specified in the command line.

You can set *X* default values for your *Emacs* windows in your *.Xdefaults* file. Use the following format:

emacs.keyword:value

where *value* specifies the default value of *keyword*. *Emacs* lets you set default values for the following keywords:

BodyFont
Sets the window's text font.

ReverseVideo
If *ReverseVideo's* value is set to *on*, the window will be displayed in inverse video.

BitMapIcon
If *BitMapIcon's* value is set to *on*, the window will iconify into the "kitchen sink."

BorderWidth
Sets the window's border width in pixels.

Foreground
For color displays, sets the window's text color. See the file */usr/lib/rgb.txt* for a list of valid *color* names.

Background
For color displays, sets the window's background color. See the file */usr/lib/rgb.txt* for a list of valid *color* names.

Border For color displays, sets the color of the window's border. See the file */usr/lib/rgb.txt* for a list of valid *color* names.

Cursor For color displays, sets the color of the window's text cursor. See the file */usr/lib/rgb.txt* for a list of valid *color* names.

Mouse For color displays, sets the color of the window's mouse cursor. See the file */usr/lib/rgb.txt* for a list of valid *color* names.

If you try to set color values while using a black and white display, the window's characteristics will default as follows: the foreground color will be set to black, the background color will be set to white, the border color will be set to grey, and the text and mouse cursors will be set to black.

Using the Mouse

The following lists the key bindings for the mouse cursor when used in an *Emacs* window.

MOUSE BUTTON	FUNCTION
left	set mark
middle	set cursor
right	select (Emacs) window
SHIFT-middle	put text into X cut buffer (cut text)
SHIFT-right	paste text
CTRL-middle	cut text and kill it

CTRL-right select this window, then split it into two windows

CTRL-SHIFT-left X buffer menu--hold the buttons and keys down, wait for menu to appear, select buffer, and release. Move mouse out of menu and release to cancel.

CTRL-SHIFT-middle X help menu--pop up index card menu for Emacs help.

CTRL-SHIFT-right Select window with mouse, and delete all other windows. Same as typing CTRL-x 1.

MANUALS

You can order printed copies of the GNU Emacs Manual for $15.00/copy postpaid from the Free Software Foundation, which develops GNU software (contact them for quantity prices on the manual). Their address is:

Free Software Foundation
675 Mass Ave.
Cambridge, MA 02139

Your local Emacs maintainer might also have copies available. As with all software and publications from FSF, everyone is permitted to make and distribute copies of the Emacs manual. The TeX source to the manual is also included in the Emacs source distribution.

FILES

/usr/src/gemacs/src - C source files and object files

/usr/lib/gemacs/lisp - Lisp source files and compiled files that define most editing commands. Some are preloaded; others are autoloaded from this directory when used.

/usr/src/gemacs/man - sources for the Emacs reference manual.

/usr/lib/emacs/etc - various programs that are used with GNU Emacs, and some files of information.

/usr/src/gemacs/etc/DOC.* - contains the documentation strings for the Lisp primitives and preloaded Lisp functions of GNU Emacs. They are stored here to reduce the size of Emacs proper.

/usr/src/gemacs/etc/DIFF discusses GNU Emacs vs. Twenex Emacs;
/usr/src/gemacs/etc/CCADIFF discusses GNU Emacs vs. CCA Emacs;
/usr/src/gemacs/etc/GOSDIFF discusses GNU Emacs vs. Gosling Emacs.
/usr/src/gemacs/etc/SERVICE lists people offering various services to assist users of GNU Emacs, including education, troubleshooting, porting and customization.
These files also have information useful to anyone wishing to write programs in the Emacs Lisp extension language, which has not yet been fully documented.

/usr/src/gemacs/info - files for the Info documentation browser (a subsystem of Emacs) to refer to. Currently not much of Unix is documented here, but the complete text of the Emacs reference manual is included in a convenient tree structured form.

/usr/lib/emacs/lock - holds lock files that are made for all files being modified in Emacs, to prevent simultaneous modification of one file by two users.

/usr/src/gemacs/cpp - the GNU cpp, needed for building Emacs on certain versions of Unix

where the standard cpp cannot handle long names for macros.

/usr/src/gemacs/shortnames - facilities for translating long names to short names in C code, needed for building Emacs on certain versions of Unix where the C compiler cannot handle long names for functions or variables.

Those files in /usr/src must be installed via the SOURCE option on the Convex Utilties distribution tape.

BUGS

There is a mailing list, bug-gnu-emacs@prep.ai.mit.edu on the internet (ucbvax!prep.ai.mit.edu!bug-gnu-emacs on UUCPnet), for reporting Emacs bugs and fixes. But before reporting something as a bug, please try to be sure that it really is a bug, not a misunderstanding or a deliberate feature. We ask you to read the section "Reporting Emacs Bugs" near the end of the reference manual (or Info system) for hints on how and when to report bugs. Also, include the version number of the Emacs you are running in *every* bug report that you send in.

Do not expect a personal answer to a bug report. The purpose of reporting bugs is to get them fixed for everyone in the next release, if possible. For personal assistance, look in the SERVICE file (see above) for a list of people who offer it.

Please do not send anything but bug reports to this mailing list. Send requests to be added to mailing lists to the special list info-gnu-emacs-request@prep.ai.mit.edu (or the corresponding UUCP address). For more information about Emacs mailing lists, see the file /usr/src/gemacs/etc/MAILINGLISTS. Bugs tend actually to be fixed if they can be isolated, so it is in your interest to report them in such a way that they can be easily reproduced.

Bugs that I know about are: shell will not work with programs running in Raw mode on some Unix versions.

UNRESTRICTIONS

Emacs is free; anyone may redistribute copies of *Emacs* to anyone under the terms stated in the *Emacs* General Public License, a copy of which accompanies each copy of *Emacs* and which also appears in the reference manual.

Copies of *Emacs* may sometimes be received packaged with distributions of Unix systems, but it is never included in the scope of any license covering those systems. Such inclusion violates the terms on which distribution is permitted. In fact, the primary purpose of the General Public License is to prohibit anyone from attaching any other restrictions to redistribution of *Emacs*.

Richard Stallman encourages you to improve and extend *Emacs,* and urges that you contribute your extensions to the GNU library. Eventually GNU (Gnu's Not Unix) will be a complete replacement for Berkeley Unix. Everyone will be able to use the GNU system for free.

AUTHORS

Emacs was written by Richard Stallman and the Free Software Foundation. Joachim Martillo and Robert Krawitz added the X features.

NAME

error – analyze and disperse compiler error messages

SYNOPSIS

error [**–n**] [**–s**] [**–q**] [**–v**] [**–T**] [**–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 the line to which 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. The **–T** option makes the output of *error* terse.

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 *fc*. *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 filename for languages that don't include the filename 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 messages 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, 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.

–T The output generated by error is terse.

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, edit, e – text editor

SYNOPSIS

ex [**–**] [**–v**] [**–t** tag] [**–r**] [**–R**] [*+command*] [**–l**] [**–x**] name ...
edit [ex options]

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*.

Options available with *ex* are:

–	Suppress some feedback from editor (prompts and messages.).
–v	Invoke the command set used in the *vi* editor.
–t	Edit the file containing the *tag* and position the editor at its definition.
–r	Recover file modifications after a system crash.
–R	Set *readonly* option which is used to read but not modify a file.
+command	Editor begins execution by processing the specified command.
–l	Sets the *showmatch* option.
–x	Editor will handle encrypted files. Note: this option is not available in the international distribution of ConvexOS.

DOCUMENTATION

The document *Edit: A tutorial* provides a comprehensive introduction to *edit* assuming no previous knowledge of computers or the ConvexOS system.

The *Ex Reference Manual – Version 3.5* 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*. All of these documents can be found in the *ConvexOS Tutorial Papers*. In addition, the *Vi Quick Reference* card summarizes the commands of *vi* in a useful, functional way, and is useful with the *Introduction*.

FILES

/usr/lib/ex?.?strings	error messages
/usr/lib/ex?.?recover	recover command
/usr/lib/ex?.?preserve	preserve command
/etc/termcap	describes capabilities of terminals
~/.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), sed(1), grep(1), vi(1), termcap(5), environ(7)

BUGS

The *undo* command causes all marks to be lost on lines changed and then restored if the marked lines were changed.

undo never clears the buffer modified condition.

The *z* command prints a number of logical rather than physical lines. More than a screen full of output may result if long lines are present.

File input/output errors don't print a name if the command line '–' option is used.

There is no easy way to do a single scan ignoring case.

The editor does not warn if text is placed in named buffers and not used before exiting the editor.

Null characters are discarded in input files, and cannot appear in resultant files.

NAME

expand, unexpand – expand tabs to spaces, and vice versa

SYNOPSIS

expand [–tabstop] [–tab1,tab2,...,tabn] [file ...]
unexpand [**–a**] [file ...]

DESCRIPTION

Expand processes the named files or the standard input, writing the standard output with tabs changed into blanks. Backspace characters are preserved into the output and decrement the column count for tab calculations. *Expand* is useful for pre-processing character files (before sorting, looking at specific columns, etc.) that contain tabs.

If a single *tabstop* argument is given then tabs are set *tabstop* spaces apart instead of the default 8. If multiple tabstops are given then the tabs are set at those specific columns.

Unexpand puts tabs back into the data from the standard input or the named files and writes the result on the standard output. By default only leading blanks and tabs are reconverted to maximal strings of tabs. If the **–a** option is given, then tabs are inserted whenever they would compress the resultant file by replacing two or more blanks before a tab stop.

NAME

explain, diction – print wordy sentences; thesaurus for diction

SYNOPSIS

diction [**–ml**] [**–mm**] [**–n**] [**–f** pfile] file ...
explain

DESCRIPTION

diction finds all sentences in a document that contain phrases from a data base of bad or wordy diction. Each phrase is bracketed with [ ]. Because *diction* runs *deroff* before looking at the text, formatting header files should be included as part of the input. The default macro package **–ms** may be overridden with the flag **–mm.** The flag **–ml** which causes **deroff** to skip lists, should be used if the document contains many lists of non-sentences. The user may supply her/his own pattern file to be used in addition to the default file with **–f** *pfile*. If the flag **–n** is also supplied the default file will be suppressed.

explain is an interactive thesaurus for the phrases found by diction.

SEE ALSO

deroff(1)

BUGS

Use of non-standard formatting macros may cause incorrect sentence breaks. In particular, *diction* doesn't understand **–me**.

NAME

expr – evaluate arguments as an expression

SYNOPSIS

expr arg ...

DESCRIPTION

The arguments are taken as an expression. After evaluation, the result is written on the standard output. Each token of the expression is a separate argument.

The operators and keywords are listed below. The list is in order of increasing precedence, with equal precedence operators grouped.

expr | *expr*
yields the first *expr* if it is neither null nor '0', otherwise yields the second *expr*.

expr & *expr*
yields the first *expr* if neither *expr* is null or '0', otherwise yields '0'.

expr relop expr
where *relop is one of* < <= = != >= >, yields '1' if the indicated comparison is true, '0' if false. The comparison is numeric if both *expr* are integers, otherwise lexicographic.

expr + *expr*
expr − *expr*
addition or subtraction of the arguments.

expr * *expr*
expr / *expr*
expr % *expr*
multiplication, division, or remainder of the arguments.

expr : *expr*
The matching operator compares the string first argument with the regular expression second argument; regular expression syntax is the same as that of *ed*(1). The \(...\) pattern symbols can be used to select a portion of the first argument. Otherwise, the matching operator yields the number of characters matched ('0' on failure).

(*expr*) parentheses for grouping.

Examples:

To add 1 to the Shell variable *a*:

```
a=`expr $a + 1`
```

To find the filename part (least significant part) of the pathname stored in variable *a*, which may or may not contain '/':

```
expr $a : '.*/\(.*\)' '|' $a
```

Note the quoted Shell metacharacters.

SEE ALSO

sh(1), test(1)

DIAGNOSTICS

Expr returns the following exit codes:

0	if the expression is neither null nor '0',
1	if the expression is null or '0',
2	for invalid expressions.

BUGS

Expr fails to handle negative arguments correctly. In particular, arithmetic operators will not accept negative arguments.

NAME

false, true – provide truth values

SYNOPSIS

true

false

DESCRIPTION

True and *false* are usually used in a Bourne shell script. They test for the appropriate status "true" or "false" before running (or failing to run) a list of commands.

SEE ALSO

csh(1), sh(1), true(1)

DIAGNOSTICS

False has exit status nonzero.

NAME

file – determine file type

SYNOPSIS

file [–**c**] [–**f** ffile] [–**m** mfile] file ...

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 the **-f** switch is used, *file* will use the contents of the next argument as a list of files to examine.

In an attempt to gather information about a given argument file (usually an object or executable), there is a table that contains byte offsets and values to check for at that offset which indicate certain qualities, such as pre-paged, not stripped, or IEEE floating point mode. The **-m** switch allows users to provide an additional magic file to search for the checks to be made against the file arguments. This user-specified file will be searched before /usr/etc/magic so that users may override the Convex-provided magic number checks.

The **-c** switch will display some debug information only: the table created from the magic file(s) will be output. All numeric values will be octal except for the entries whose opcode field is MSK; in that case, numbers will be in hex.

FILES

/usr/etc/magic table of magic numbers and other checks to perform on files

SEE ALSO

magic(5)

BUGS

It often makes mistakes. In particular it often suggests that command files are C programs.

NAME

find – find files

SYNOPSIS

find *pathname-list expression*

DESCRIPTION

find recursively descends the directory hierarchy for each pathname in the *pathname-list* (i.e., one or more pathnames) seeking files that match a boolean *expression* written in the primaries given below. In the descriptions, the argument *n* is used as a decimal integer where *+n* means more than *n*, *-n* means less than *n*, and *n* means exactly *n*.

–name *filename*
True if the *filename* argument 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, f, l, p** or **s** for block special file, character special file, directory, plain file, symbolic link, named pipe (FIFO), or socket.

-fstype *type*
True if the filesystem to which the file belongs is of type *type*, where *type* is typically **4.2** or **nfs**

–links *n* True if the file has *n* links.

–user *uname*
True if the file belongs to the user *uname* (login name or numeric user ID).

–nouser True if the file belongs to a user *not* in the */etc/passwd* database.

–group *gname*
True if the file belongs to group *gname* (group name or numeric group ID).

–nogroup True if the file belongs to a group *not* in the */etc/group* database.

–size *n* True if the file is *n* blocks long (512 bytes per block).

–inum *n* True if the file has inode number *n*.

–atime *n* True if the file has been accessed in the *n*th previous 24-hour period. The 0th 24-hour period is the time between the current time and 24 hours before, the 1st 24-hour period is the time between 24 and 48 hour ago, and so on.

–mtime *n* True if the file has been modified in the *n*th previous 24-hour period. The 0th 24-hour period is the time between the current time and 24 hours before, the 1st 24-hour period is the time between 24 and 48 hour ago, and so on.

–exec *command*
True if the executed command returns a zero value as exit status. The end of the command must be punctuated by an escaped semicolon. A command argument (**{}**) is replaced by the current pathname.

–ok *command*
Like **–exec** except that the generated command is written on the standard output, then the standard input is read and the command executed only upon response **y**.

–mig True if any portion of the file is not resident on disk.

–print Always true; causes the current pathname to be printed.

–ls Always true; causes current pathname to be printed together with its associated statistics. These include (respectively) inode number, size in kilobytes (1024 bytes), protection mode, number of hard links, user, group, size in bytes, and modification time. If the file is a special file the size field will instead contain the major and minor device numbers. If the file is a symbolic link the pathname of the linked-to file is printed

preceded by "->". The format is identical to that of "ls -gilds" (note however that formatting is done internally, without executing the ls program).

-newer *file*
True if the current file has been modified more recently than the argument *file*.

-prune Always returns true. Has the side effect of pruning the search tree at the file. That is, if the current path name is a directory, *find* will not descend into that directory.

-xdev Always true; causes *find* **not** to traverse down into a file system different from the one on which current *argument* pathname resides.

-cpio *file* True if cpio returns a zero value as exit status. The current pathname is appended to file using cpio.

The primaries may be combined using the following operators (in order of decreasing precedence):

1) A parenthesized group of primaries and operators (parentheses are special to the Shell and must be escaped).
2) The negation of a primary ("!" is the unary *not* operator).
3) Concatenation of primaries (the *and* operation is implied by the juxtaposition of two primaries).
4) Alternation of primaries (**-o** is the *or* operator).

EXAMPLE

To remove all files named "a.out" or "*.o" that have not been accessed for a week:

find / \(-name a.out -o -name '*.o' \) -atime +7 -exec rm {} \;

To remove all files name "a.out" or "*.o" that reside on an nfs mounted partition:

find / \(-name a.out -o -name '*.o' \) -fstype nfs -exec rm {} \;

NOTE

When descending a directory hierarchy, if *find* encounters a symbolic link to a directory, it will not follow the link. This means that files in the linked-to directory will not be detected by *find*.

find should not be used on nfs filesystems unless absolutely necessary. Such use will cause the NFS servers to thrash. Use the *-fstype 4.2* argument in prevent delving into nfs filesystems.

EXIT STATUS

If *find* does not find any matches to its parameters, it will exit with a status of 101.

FILES

/etc/passwd
/etc/group

find is able to find files of greater than two gigabytes, and print out the information using the **-ls** option.

SEE ALSO

sh(1), test(1), fs(5)

BUGS

The syntax is cumbersome.

NAME

finger – user information lookup program

SYNOPSIS

finger [**–bfhilmpqsw**] [login1 [login2 ...]]

DESCRIPTION

By default, *finger* lists the login name, full name, terminal name and write status (as an "*" before the terminal name if write permission is denied), idle time, login time, and office location and phone number (if they are known) for each current user. (Idle time is expressed in minutes if it is a single integer; hours and minutes if a ":" is present; or days and hours if a "d" is present.)

A longer format also exists and is used by *finger* whenever a list of people's names is given. (Account names as well as first and last names of users are accepted.) This format is multi-line, and includes all the information described above as well as the user's home directory and login shell; the content of the user's *.plan* file in the user's home directory; and the contents of the user's *.project* file, also located in the user's home directory.

finger may be used to look up users on a remote machine by specifying the user as "user@host." If the user name is omitted, the standard format listing is provided on the remote machine.

finger options include:

–m Match arguments only on user name.

–l Force long output format.

–b Use briefer long output format.

–q Quick list of users. (Looks similar to *who*(1) output.)

–i Quick list of users with idle time.

–s *name* [...] Force short output format.

–w *name* [...] Force narrow short output format.

–f Suppress printing of the headers for short and quick outputs.

–p Suppress printing of the *.plan* files.

–h Suppress printing of the *.project* files.

Short form output normally contains the office location and the office phone number. If a user is logged in on a dialup line, *finger* assumes that the user is not in the office. In this case, the office field is left blank and the home phone number is listed in the phone field.

FILES

/etc/utmp	who file
/etc/passwd	for users names, offices, ...
/usr/adm/lastlog	last login times
~/.plan	plans
~/.project	projects

SEE ALSO

chfn(1), w(1), who(1), passwd(5), ttys(5), utmp(5)

BUGS

Because *finger* uses an internet standard port, there is no way to pass arguments to the remote machine.

NAME

fmt – simple text formatter

SYNOPSIS

fmt [–width] [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 *width* characters long as possible (72, if *width* is not specified). The spacing at the beginning of the input lines is preserved in the output, as are blank lines and interword spacing.

Fmt is meant to format mail messages prior to sending, but may also be useful for other simple tasks. For instance, within visual mode of the *ex* editor (e.g. *vi*) the command

```
%!fmt
```

will reformat a paragraph, evening the lines.

SEE ALSO

nroff(1), mail(1)

AUTHOR

Kurt Shoens

BUGS

The program was designed to be simple and fast – for more complex operations, the standard text processors are likely to be more appropriate.

NAME

fold – fold long lines for finite width output device

SYNOPSIS

fold [–width] [file ...]

DESCRIPTION

Fold is a filter which will fold the contents of the specified files, or the standard input if no files are specified, breaking the lines to have maximum width *width*. The default for *width* is 80. *Width* should be a multiple of 8 if tabs are present, or the tabs should be expanded using *expand*(1) before coming to *fold*.

SEE ALSO

expand(1)

BUGS

If underlining is present it may be messed up by folding.

NAME

from – who is my mail from?

SYNOPSIS

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

biff(1), mail(1), prmail(1)

NAME

ftp – ARPANET file transfer program

SYNOPSIS

ftp [**–v**] [**–d**] [**–i**] [**–n**] [**–g**] [**–u**] [*host*]

DESCRIPTION

ftp is the user interface to the ARPANET standard File Transfer Protocol (FTP). 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* immediately attempts to establish a connection to an FTP server on that host; otherwise, *ftp* enters its command interpreter and awaits instructions from the user. While *ftp* is awaiting commands, it displays the prompt, **ftp>**. The following commands are recognized by *ftp*:

! [*command* [*args*]]
Invoke an interactive shell on the local machine. If there are arguments, the first one is interpreted as the command; the rest serve as arguments for that command.

$ *macro-name* [*args*]
Execute the macro *macro-name* that was defined with the *macdef* command. Arguments are passed to the macro unglobbed.

account [*password*]
Supply a supplemental password required by a remote system for access to resources once a login has been successfully completed. If no argument is included, the user is prompted for an account password in a non-echoing input mode.

append *local-file* [*remote-file*]
Append a local file to a file on the remote machine. If *remote-file* is not specified, the local file name is used to name the remote file after being altered by any *ntrans* or *nmap* settings. File transfer uses the current settings for *type*, *format*, *mode*, and *structure*.

ascii Set the file transfer *type* to network ASCII, the default *type*.

bell Sound a bell after each file transfer command completes.

binary Set the file transfer *type* to support binary image transfer.

bye Terminate the session with the remote FTP server and exit *ftp*. An end of file also terminates the session and causes *ftp* to exit.

case Toggle remote computer file name case mapping during *mget* commands. When *case* is on (default is off), remote computer file names with all letters in upper case are written in the local directory in lower case.

cd *remote-directory* Change the working directory on the remote machine to *remote-directory*.

cdup Change the remote machine working directory to the parent of the current remote machine working directory.

close Terminate the FTP session with the remote server, and return to the command interpreter. Any defined macros are erased.

cr Toggle carriage return stripping during ASCII *type* file retrieval. Records are delimited by a carriage return/linefeed sequence during ASCII *type* file transfer. When *cr* is on (the default), carriage returns are stripped from this sequence to conform with the UNIX single linefeed record delimiter. Records on non-UNIX remote systems may contain single linefeeds; when an ASCII *type* transfer is made, these linefeeds may be distinguished from a record delimiter only when *cr* is off. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

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 contents in the *remote-directory*, and optionally place 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, or if the *local-file* is "-," output is sent to the terminal. If interactive prompting is on, *ftp* prompts the user to verify that the last argument is indeed the target local file for receiving *dir* output.

disconnect
A synonym for *close*.

form *format*
Set the file transfer *form* to *format*. The default format is **file**.

get *remote-file* [*local-file*]
Retrieve *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, subject to alteration by the current *case*, *ntrans*, and *nmap* settings. The current settings for *type*, *form*, *mode*, and *structure* are used while transferring the file.

glob Toggle file name expansion for *mdelete*, *mget*, and *mput*. If globbing is turned off with *glob*, the file name arguments are taken literally and not expanded. Globbing for *mput* is done as in *csh*(1). For *mdelete* and *mget*, each remote file name is expanded separately on the remote machine and the lists are not merged. Expansion of a directory name is likely to be different from expansion of the name of an ordinary file; the exact result depends on the foreign operating system and FTP server, and can be previewed by entering **mls** *remote-files* **-**. Note: *mget* and *mput* are not meant to transfer entire directory subtrees of files. That can be done by transferring a *tar*(1) archive of the subtree (in binary mode).

hash Toggle hash-sign ("#") printing for each data block transferred. The size of a data block is 1024 bytes.

help [*command*]
Print an informative message about the meaning of *command*. If no argument is given, *ftp* prints a list of 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 a listing of the contents of a directory on the remote machine. The listing includes any system-dependent information that the server chooses to include; for example, most UNIX systems will produce output from the command **ls -l**. (See also **nlist**.) If *remote-directory* is not specified, the current working directory is used. If interactive prompting is on, *ftp* prompts the user to verify that the last argument is indeed the target local file for receiving *ls* output. If no local file is specified, or *local-file* is "-," the output is sent to the terminal.

macdef *macro-name*
Define a macro. Subsequent lines are stored as the macro *macro-name*; a null line (consecutive newline characters in a file or carriage returns from the terminal) terminates

macro input mode. There is a limit of 16 macros and 4096 total characters in all defined macros. Macros remain defined until a *close* command is executed. The macro processor interprets "$" and "\" as special characters. A "$" followed by a number (or numbers) is replaced by the corresponding argument on the macro invocation command line. A "$" followed by an "i" signals the macro processor that the executing macro is to be looped. On the first pass, "$i" is replaced by the first argument on the macro invocation command line; on the second pass, it is replaced by the second argument, and so on. A "\" followed by any character is replaced by that character. Use the "\" to prevent special treatment of the "$."

mdelete [*remote-files*]
Delete *remote-files* on the remote machine.

mdir *remote-files local-file*
Functions like *dir*, except that multiple remote files may be specified. If interactive prompting is on, *ftp* prompts the user to verify that the last argument is indeed the target local file for receiving *mdir* output.

mget *remote-files*
Expand file names for *remote-files* on the remote machine and do a *get* for each file name produced. See *glob* for details on file name expansion. Resulting file names are then processed according to *case*, *ntrans*, and *nmap* settings. Files are transferred into the local working directory, which can be changed with **lcd** *directory*; new local directories can be created with **! mkdir** *directory*.

mkdir *directory-name*
Make a directory on the remote machine.

mls *remote-files local-file*
Like **nlist**, except multiple remote files may be specified, and the *local-file* must be specified. If interactive prompting is on, *ftp* will prompt the user to verify that the last argument is indeed the target local file for receiving **mls** output.

mode [*mode-name*]
Set the file transfer *mode* to *mode-name*. The default mode is **stream** mode.

modtime *file-name*
Show the last modification time of the file on the remote machine.

mput *local-files*
Expand wild cards in the list of local files given as arguments and do a *put* for each file in the resulting list. See *glob* for details on file name expansion. Resulting file names are then processed according to *ntrans* and *nmap* settings.

nlist [*remote-directory*] [*local-file*]
Print a list of the files in a directory on the remote machine. If *remote-directory* is left unspecified, the current working directory is used. If interactive prompting is on, *ftp* will prompt the user to verify that the last argument is indeed the target local file for receiving *nlist* output. If no local file is specified, or if *local-file* is -, the output is sent to the terminal.

nmap [*inpattern outpattern*]
Set or reset the file name mapping mechanism. If no arguments are specified, the file name mapping mechanism is reset. If arguments are specified, remote file names are mapped during *mget* commands and *get* commands issued without a specified remote target file name, and local file names are mapped during *mget* commands and *get* commands issued without a specified local target file name. This command is useful when connecting to a non-UNIX remote computer with different file naming conventions or practices.

The mapping follows the pattern set by *inpattern* and *outpattern*. The *inpattern* is a template for incoming file names (which may have already been processed according to the *ntrans* and *case* settings). Variable templating is accomplished by including the sequences "$1," "$2," *inpattern*. Use "\" to prevent special treatment of the "$" character. All other characters are treated literally, and are used to determine the *nmap inpattern* variable values. For example, given *inpattern* "$1.$2" and the remote file name "mydata.data," $1 would have the value "mydata," and $2 would have the value "data."

The *outpattern* determines the resulting mapped file name. The sequences "$1," "$2," ..., "$9" are replaced by any value resulting from the *inpattern* template. The sequence "$0" is replaced by the original file name. Additionally, the sequence "[*seq1*,*seq2*]" is replaced by *seq1* if *seq1* is not a null string; otherwise it is replaced by *seq2*. For example, the command **nmap $1.$2.$3 [$1,$2].[$2,file]** would yield the output file name "myfile.data" for input file names "myfile.data" and "myfile.data.old," "myfile.file" for the input file name "myfile," and "myfile.myfile" for the input file name ".myfile." Spaces may be included in *outpattern*, as in the example: **nmap $1 |sed "s/ *$//" > $1**. Use the "\" character to prevent special treatment of the "$," "[," "]," and "," characters.

ntrans [*inchars* [*outchars*]]
Set or reset the file name character translation mechanism. If no arguments are specified, the file name character translation mechanism is reset. If arguments are specified, characters in remote file names are translated during *mput* commands and *put* commands issued without a specified remote target file name, and characters in local file names are translated during *mget* commands and *get* commands issued without a specified local target file name. This command is useful when connecting to a non-UNIX remote computer with different file naming conventions or practices. Characters in a file name matching a character in *inchars* are replaced with the corresponding character in *outchars*. If the character's position in *inchars* is longer than the length of *outchars*, the character is deleted from the file name.

open *host* [*port*]
Establish a connection to the specified *host* FTP server. An optional port number may be supplied, in which case, *ftp* attempts to contact an FTP server at that port. If the *autologin* option is on (the default), *ftp* also attempts 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 is on), any *mget* or *mput* command transfers all files, and any *mdelete* command deletes all files.

proxy *ftp-command*
Execute an *ftp* command on a secondary control connection. This command allows simultaneous connections to two remote FTP servers for transferring files between the two servers. The first *proxy* command should be an *open*, to establish the secondary control connection. Enter the command, **proxy ?**, to see other *ftp* commands executable on the secondary connection.

The following commands behave differently when prefaced by *proxy*: *open* does not define new macros during the auto-login process; *close* does not erase existing macro definitions; *get* and *mget* transfer files from the host on the primary control connection to the host on the secondary control connection, and *put*, *mput*, and *append* transfer files from the host on the secondary control connection to the host on the primary control

connection. Third party file transfers depend upon support of the FTP protocol PASV command by the server on the secondary control connection.

put *local-file* [*remote-file*]
Store a local file on the remote machine. If *remote-file* is not specified, the local file name is used after processing according to any *ntrans* or *nmap* settings 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.

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.

remotestatus [*file-name*]
With no arguments, show status of remote machine. If *file-name* is specified, show status of *file-name* on remote machine.

rename [*from*] [*to*]
Rename the file *from* to the file *to* on the remote machine.

reset Clear reply queue. This command re-synchronizes command/reply sequencing with the remote FTP server. Resynchronization may be necessary following a violation of the FTP protocol by the remote server.

rmdir *directory-name* Delete a directory on the remote machine.

runique Toggle storing of files on the local system with unique file names. If a file already exists with a name equal to the target local file name for a *get* or *mget* command, a "**.1**" is appended to the name. If the resulting name matches another existing file, a "**.2**" is appended to the original name. If this process continues up to ".99," an error message is printed, and the transfer does not take place. The generated unique file name is reported. Note that **runique** will not affect local files generated from a shell command (see below). The default value is off.

send *local-file* [*remote-file*]
A synonym for *put*.

sendport
Toggle the use of PORT commands. By default, *ftp* attempts to use a PORT command when establishing a connection for each data transfer. The use of PORT commands can prevent delays when performing multiple file transfers. If the PORT command fails, *ftp* uses the default data port. When the use of PORT commands is disabled, no attempt is made to use PORT commands for each data transfer. This is useful for certain FTP implementations that do ignore PORT commands but, incorrectly indicate that they have been accepted.

size *file-name* Return size of *file-name* on remote machine.

status Show the current status of *ftp*.

struct [*struct-name*]
Set the file transfer *structure* to *struct-name*. By default, "**streamR" structure is used.**

sunique
Toggle storing of files on remote machine under unique file names. Remote FTP server

must support the FTP protocol STOU command for successful completion. The remote server reports the unique name. Default value is off.

system
Show the type of operating system running on the remote machine.

tenex Set the file transfer *type* to that needed to talk to TENEX machines.

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* prompts the user for it (after disabling local echo). If an *account* field is not specified, and the FTP server requires it, the user is prompted for it. If an *account* field is specified, an account command is relayed to the remote server after the login sequence completes, if the remote server did not require it for logging in. 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. In addition, if *verbose* is on, when a file transfer completes, statistics regarding the efficiency of the transfer are reported. By default, *verbose* is on.

? [*command*]
A synonym for *help*.

Command arguments that have embedded spaces may be quoted with quote (") marks.

ABORTING A FILE TRANSFER

To abort a file transfer, use the terminal interrupt key (usually <CTRL-C>). Sending transfers are immediately halted. Receiving transfers are halted by sending an FTP protocol ABOR command to the remote server, and discarding any further data received. The speed at which this is accomplished depends upon the remote server's support for ABOR processing. If the remote server does not support the ABOR command, an **ftp>** prompt will not appear until the remote server has completed sending the requested file.

The terminal interrupt key sequence is ignored when *ftp* has completed any local processing and is awaiting a reply from the remote server. A long delay in this mode may result from the ABOR processing described above, or from unexpected behavior by the remote server, including violations of the FTP protocol. If the delay results from unexpected remote server behavior, the local *ftp* program must be killed by hand.

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 (writes) from the stdout (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 used in the *csh*(1); c.f. the *glob* command. If the *ftp* command expects a single local file (e.g., *put*), only the first file name generated by the "globbing" operation is used.

4) For *mget* commands and *get* commands with unspecified local file names, the local file

name is the remote file name, which may be altered by a *case*, *ntrans*, or *nmap* setting. The resulting file name may then be altered if *runique* is on.

5) For *mput* commands and *put* commands with unspecified remote file names, the remote file name is the local file name, which may be altered by a *ntrans* or *nmap* setting. The resulting file name may then be altered by the remote server if *sunique* is on.

FILE TRANSFER PARAMETERS

The FTP specification specifies many parameters that may affect a file transfer. The *type* may be one of ASCII, image (binary), ebcdic, and local byte size (for PDP-10s and PDP-20s, mostly). *ftp* supports the ASCII and image types of file transfer, plus local byte size 8 for *tenex* mode transfers.

ftp supports only the default values for the remaining file transfer parameters: *mode*, *form*, and *struct*.

OPTIONS

Options may be specified on the command line, or entered with 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* checks the *.netrc* (see below) file in the user's home directory for an entry describing an account on the remote machine. If no entry exists, *ftp* prompts the user for the remote machine login name (default is the user identity on the local machine) and, if necessary, prompts for a password and an account with which to log in.

The **–i** option turns off interactive prompting during multiple file transfers.

The **–d** option enables debugging.

The **–g** option disables file name globbing.

THE *.netrc* FILE

The *.netrc* file contains login and initialization information used by the auto-login process. It resides in the user's home directory. The following tokens are recognized; they may be separated by spaces, tabs, or new-lines.

machine *name*

Identify a remote machine name. The auto-login process searches the *.netrc* file for a *machine* token that matches the remote machine specified on the *ftp* command line or as an *open* command argument. Once a match is found, the subsequent *.netrc* tokens are processed, stopping when the end of file is reached or another *machine* token is encountered.

default

This is the same as *machine name except that default matches any name. There can be only one default token, and it must be after all machine tokens. This is normally used as:*

default *login anonymous password user@site* thereby giving the user, *automatic*, anonymous *ftp* login to machines not specified in *.netrc*. This can be overridden by using the **–n** flag to disable auto-login.

login *name*

Identify a user on the remote machine. If this token is present, the auto-login process initiates a login using the specified name.

password *string*

Supply a password. If this token is present, the auto-login process supplies the specified string if the remote server requires a password as part of the login process. If this token is present in the *.netrc* file, *ftp* aborts the auto-login process if the *.netrc* is readable by anyone besides the user.

account *string*

Supply an additional account password. If this token is present, the auto-login process supplies the specified string if the remote server requires an additional account password, or the auto-login process initiates an ACCT command if it does not.

macdef *name*

Define a macro. This token functions like the *ftp macdef* command. A macro is defined with the specified name; its contents begin with the next *.netrc* line and continue until a null line (consecutive new-line characters) is encountered. If a macro named *init* is defined, it is automatically executed as the last step in the auto-login process.

SEE ALSO

ftpd(8C)

BUGS

Correct execution of many commands depends upon proper behavior by the remote server.

An error in the treatment of carriage returns in the 4.2BSD UNIX ASCII mode transfer code has been corrected. This correction may result in incorrect transfers of binary files to and from 4.2BSD servers using the **ascii** type. Avoid this problem by using the **binary** image *type*.

NOTES

ftp understands that files on CONVEX systems may be greater than two gigabytes in size. This may cause problems with other vendors' implementation of *ftp*.

ftp is an optional product; for more information, contact your CONVEX sales representative.

NAME

getsysinfo – print out system information

SYNOPSIS

getsysinfo [**–v**] [**–f***feature* | **–l**]
cpuid [**–v**]
arch [**–v**]

DESCRIPTION

getsysinfo prints configuration and feature information about the system. It returns information that is available via the *getsysinfo*(2) system call. This information includes CPU serial number, type of the machine (e.g., C1, C200), type of floating point hardware that is available and its default mode at boot time, and whether certain machine instructions exist. *arch*, which is provided for Sun compatibility, and *cpuid* are just special cases of, and links to, *getsysinfo*.

arch is equivalent to

getsysinfo -f cpu_type

and *cpuid* is equivalent to

getsysinfo -f system_sn

In fact, if *getsysinfo* is called by a name other than these three, assumes its name is the name of a **feature** that is to be printed. For example, creating a symbolic link to *getsysinfo* called "cpu_count", then executing *getsysinfo* via the new link will just print out the number of configured processors. Executing via this new link is equivalent to

getsysinfo -f cpu_count

OPTIONS

getsysinfo accepts the following options:

–v puts *getsysinfo, arch,* and *cpuid* into verbose mode. As expected, all three print additional information in verbose mode.

–f *feature* used by shell scripts that need to either test for the existence of some feature or require the value of some feature. This option must be followed by the name of a known *feature* . If the feature named is a boolean test (such as default floating-point mode or special machine instructions) *getsysinfo* will exit with a zero value if the feature exists on the machine; otherwise it will exit with a nonzero value. In either case it prints nothing. If the feature has an output value, such as system serial number or number of configured processors, *getsysinfo* will print only the value and exit with a zero status.

–l lists all known features without regard to the configuration of the machine. The options **–l** and **–f** are mutually exclusive.

FEATURES

Although possibly incomplete, this is a listing of the features that *getsysinfo* knows about. For a complete list, use the command

getsysinfo -l

or

getsysinfo -l -v

The name of each feature comes from the include file */usr/include/sys/utsname.h*. Many of the values listed here are set in SPU configuration files.

mem_interleave_factor
is the current memory interleave factor. The concept of memory interleaving is discussed in section 7.9 of the *CONVEX Architecture Reference.*

native_default and **ieee_default**
are mutually exclusive boolean values. They refer to the default representation of floating-point numbers.

ieee_supported
is a boolean value and is true if IEEE floating-point hardware is available. Native and IEEE floating-point representations are discussed in section 2.5 of the *CONVEX Architecture Reference.*

system_sn

is the serial number of the machine.

cpu_type is the exact name of the machine architecture. Chapter 7 of the *CONVEX Architecture Reference* discusses the differences between architectures.

cpu_count
is the number of processors installed in the system.

parallel is a boolean value and is true if parallel processing instructions are available on the machine. Chapter 5 of the *CONVEX Architecture Reference* discusses multiple processors and parallel processing.

intrinsics is a boolean value and is true if the intrinsic instructions are available on the machine. Section 8.5 of the *CONVEX Architecture Reference* discusses intrinsic instructions.

op_under_mask
is a boolean value and is true if the architecture supports operating vector/scalar instructions under mask. Section 8.10.2 of the *CONVEX Architecture Reference* discusses this feature.

scalar_acc
is a boolean value and is true if the machine has optional hardware that accelerates scalar operations.

secure_ucode
is a boolean value and is true if the secure mode of the microcode has been enabled.

WARNINGS

The "Number of configured processors" returned is just that: it is not necessarily the number of processors that are enabled. However, normally the number of processors configured is the same as the number of processors enabled. Use the *cpuconf* command to get the true number of active processors.

DIAGNOSTICS

Incorrect arguments or usage result in an error message, usage message, and an exit status of *EX_USAGE*. If the *getsysinfo (2)* system call fails, *getsysinfo (1)* will exit with an exit status of *EX_OSERR*. If the **-f** option is not used and no errors are encountered, *getsysinfo* will exit with a zero status. See */usr/include/sysexits.h* for more information on system exit statuses.

SEE ALSO

getsysinfo(2), syspic(8), cpuconf(8)

NAME

graph – draw a graph

SYNOPSIS

graph [option] ...

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 *plot*(1G) filters.

If the coordinates of a point are followed by a nonnumeric 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 newlines.

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.

–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

plot(1G), spline(1G)

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

grep, egrep, fgrep – search a file for a pattern

SYNOPSIS

grep [option] ... expression [file] ...

egrep [option] ... [expression] [file] ...

fgrep [option] ... [strings] [file] ...

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 *ex*(1); it uses a compact nondeterministic algorithm. *Egrep* patterns are full regular expressions; it uses a fast deterministic algorithm that sometimes needs exponential space. *Fgrep* patterns are fixed strings. The following options are recognized.

–v All lines but those matching are printed.

–x (Exact) only lines matched in their entirety are printed (*fgrep* only).

–c Only a count of matching lines is printed.

–l The names of files with matching lines are listed (once) separated by newlines.

–n Each line is preceded by its relative line number in the file.

–b Each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.

–h Suppress the printing of file names on multiple file greps.

–i The case of letters is ignored in making comparisons — that is, upper and lower case are considered identical (except characters enclosed by brackets [] -- see below). This applies to *grep* and *fgrep* only.

–y Equivalent to the **–i** switch.

–s Silent mode. Nothing is printed (except error messages). This is useful for checking the error status.

–w The expression is searched for as a word (as if surrounded by '\<' and '\>', see *ex*(1).) (*grep* only)

–e *expression*
Same as a simple *expression* argument, but useful when the *expression* begins with a –.

–f *file* The regular expression (*egrep*) or string list (*fgrep*) is taken from the *file*.

In all cases the file name is shown if there is more than one input file. Care should be taken when using the characters $ * [^ | ( ) and \ in the *expression* as they are also meaningful to the Shell. It is safest to enclose the entire *expression* argument in single quotes ' '.

Fgrep searches for lines that contain one of the (newline-separated) *strings*.

Egrep accepts extended regular expressions. In the following description 'character' excludes newline:

A \ followed by a single character other than newline matches that character.

The character ^ matches the beginning of a line.

The character $ matches the end of a line.

A . (period) matches any character.

A single character not otherwise endowed with special meaning matches that character.

A string enclosed in brackets [] matches any single character from the string. This match is always case sensitive, and cannot be overriden by the -i flag. Ranges of ASCII character codes may be abbreviated as in 'a–z0–9'. A] may occur only as the first character of the string. A literal – must be placed where it can't be mistaken as a range indicator.

A regular expression followed by an * (asterisk) matches a sequence of 0 or more matches of the regular expression. A regular expression followed by a + (plus) matches a sequence

of **1** or more matches of the regular expression. A regular expression followed by a ? (question mark) matches a sequence of **0** or **1** matches of the regular expression.

Two regular expressions concatenated match a match of the first followed by a match of the second.

Two regular expressions separated by | or newline match either a match for the first or a match for the second.

A regular expression enclosed in parentheses matches a match for the regular expression.

The order of precedence of operators at the same parenthesis level is [] then *+? then concatenation then | and newline.

Ideally there should be only one *grep*, but we don't know a single algorithm that spans a wide enough range of space-time tradeoffs.

NOTES

The *grep* utility can be used on large files by piping the contents of the large file to the *grep* command using *cat* (e.g. cat largefile | grep xxx).

SEE ALSO

ex(1), sed(1), sh(1)

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files.

BUGS

Lines are limited to 256 characters; longer lines are truncated.

egrep gets confused when reading characters with the most significant bit set and may fail to output all occurrences of the specified pattern.

NAME

groups – show group memberships

SYNOPSIS

groups [user]

DESCRIPTION

The *groups* command shows the groups to which you or the optionally specified user belong. Each user belongs to a group specified in the password file */etc/passwd* and possibly to other groups as specified in the file */etc/group*. If you do not own a file but belong to the group which it is owned by then you are granted group access to the file.

When a new file is created it is given the group of the containing directory.

SEE ALSO

setgroups(2)

FILES

/etc/passwd, /etc/group

BUGS

More groups should be allowed.

NAME

gut – remove text/data/bss sections from an executable

SYNOPSIS

gut

DESCRIPTION

gut reads standard input, expecting to find a **SOFF** executable. It writes a new **SOFF** executable that has zero-length text/data/bss sections to standard output. The new executable is basically an executable with only a symbol table.

gut is essentially an inverse operation to *strip*(1).

gut is intended for saving disk space with */vmunix* – instead of copying it from the **SPU** with

```
spu -r /mnt/os/vmunix > /vmunix
```

it would be copied with

```
spu -r /mnt/os/vmunix | gut > /vmunix
```

This command is typically found in */etc/rc.local*.

SEE ALSO

ld(1), strip(1)

NAME

head – give first few lines

SYNOPSIS

head [–count] [file ...]

DESCRIPTION

This filter gives the first *count* lines of each of the specified files, or of the standard input. If *count* is omitted it defaults to 10.

SEE ALSO

tail(1)

NAME

help – online information system

SYNOPSIS

help

DESCRIPTION

Help is a menu driven online information system intended for novice ConvexOS users. It provides descriptions of ConvexOS commands with examples. It is meant to be easier to use and somewhat less confusing than the ConvexOS man pages.

SEE ALSO

info(1)

NAME

hostid – set or print identifier of current host system

SYNOPSIS

hostid [hexnum or internet address]

DESCRIPTION

hostid prints the identifier of the current host in hexadecimal. This numeric value is expected to be unique across all hosts and is commonly set to the host's Internet address. The superuser can set the hostid by giving a hexadecimal argument or the hostname; this is usually done in the startup script, */etc/rc.local*.

SEE ALSO

gethostid(2)

NAME

hostname - set or print name of current host system

SYNOPSIS

hostname [**-s**] [*nameofhost*]

DESCRIPTION

hostname prints the name of the current host. The super-user can set the hostname by supplying an argument; this is usually done in the startup script script */etc/rc.local*, normally run at boot time. The *-s* option trims any domain information from the printed name.

SEE ALSO

gethostname(2), sethostname(2)

NAME

ident – identify files

SYNOPSIS

ident file ...

DESCRIPTION

Ident searches the named files for all occurrences of the pattern $*keyword:...*$, where *keyword* is one of

Author
Date
Header
Locker
Log
Revision
Source
State

These patterns are normally inserted automatically by the RCS command *co (1)*, but can also be inserted manually.

Ident works on text files as well as object files. For example, if the C program in file f.c contains

```
char rcsid[] = "$Header: Header information $";
```

and f.c is compiled into f.o, then the command

```
ident f.c f.o
```

will print

```
f.c:
     $Header: Header information $
f.o:
     $Header: Header information $
```

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 0.7 ; Release Date: 91/10/08 .

SEE ALSO

ci (1), co (1), rcs (1), rcsdiff(1), rcsmerge (1), rlog (1), rcsfile (5), sccstorcs (1).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

NAME

idtoname – numeric ID to name filter

SYNOPSIS

idtoname -[**aegtux**]...

DESCRIPTION

idtoname takes its standard input, performs field conversions as specified by its arguments, and writes to its standard output. Fields are separated by spaces.

Each option represents one field. If a line contains more fields than options, the excess part of the line is echoed. If a line contains a field whose type does not agree with the type given by the corresponding command-line argument, the entire line is echoed unchanged. *Awk*(1) may be used to format data before or after it is sent through **idtoname**.

The options are:

a interpret the field as an activity ID; convert it to an activity name.

e echo the field without modification.

g interpret the field as a group ID; convert it to a group name.

t interpret the field as a time stamp as returned by *time*(3); convert it to a 17-character ASCII string of the form "Feb 25 1986 20:36".

u interpret the field as a user ID; convert it to a user name.

x do not print the field.

EXAMPLE

Use the line printer accounting file to format the output as follows: convert the first field to a date/time, echo the second field without modification, convert the third field to a user name, the fourth field to a group name, the fifth field to activity name, remove the sixth and seventh fields, and echo the remainder of the line:

idtoname -teugaxx < /usr/adm/lpd-acct

SEE ALSO

awk(1), activities(5), group(5), passwd(5),
"Accounting" chapter in the *CONVEX System Manager's Guide.*

BUGS

idtoname does a linear search to find IDs in the appropriate data file, so the search is not as fast as it could be.

NAME

indent – indent and format C program source

SYNOPSIS

indent [*input-file* [*output-file*]] [**–bad** | **–nbad**] [**–bap** | **–nbap**] [**–bbb** | **–nbbb**] [**–bc** | **–nbc**] [**–bl** | **–br**] [**–***cn*] [**–***cdn*] [**–cdb** | **–ncdb**] [**–ce** | **–nce**] [**–***cin*] [**–***clin*] [**–***dn*] [**–***din*] [**–dj** | **–ndj**] [**–ei** | **–nei**] [**–fc1** | **–nfc1**] [**–fcd** | **–nfcd**] [**–***in*] [**–ip** | **–nip**] [**–***ln*] [**–***lcn*] [**–lp** | **–nlp**] [**–npro**] [**–pcs** | **–npcs**] [**–ps** | **–nps**] [**–psl** | **–npsl**] [**–sc** | **–nsc**] [**–sob** | **–nsob**] [**–st**] [**–troff**] [**–v** | **–nv**]

DESCRIPTION

indent is a **C** program formatter. It reformats the **C** program in the *input-file* according to the switches. The switches which can be specified are described below. They may appear before or after the file names.

NOTE: If you only specify an *input-file*, the formatting is done 'in-place', that is, the formatted file is written back into *input-file* and a backup copy of *input-file* is written in the current directory. If *input-file* is named '/blah/blah/file', the backup file is named file.*BAK*. If no *input-file* is specified, the **C** program is read from standard input and the output of *indent* is directed to standard output.

If *output-file* is specified, *indent* checks to make sure it is different from *input-file*.

OPTIONS

The options listed below control the formatting style imposed by *indent*.

–bad,–nbad If **–bad** is specified, a blank line is forced after every block of declarations. Default: **–nbad.**

–bap,–nbap If **–bap** is specified, a blank line is forced after every procedure body. Default: **–nbap.**

–bbb,–nbbb If **–bbb** is specified, a blank line is forced before every block comment. Default: **–nbbb.**

–bc,–nbc If **–bc** is specified, then a newline is forced after each comma in a declaration. **–nbc** turns off this option. The default is **–nbc.**

–br,–bl Specifying **–bl** lines up compound statements like this:

```
if (...)
{
    code
}
```

Specifying **–br** (the default) makes them look like this:

```
if (...) {
    code
}
```

–c*n* The column in which comments on code start. The default is 33.

–cd*n* The column in which comments on declarations start. The default is for these comments to start in the same column as those on code.

–cdb,–ncdb Enables (disables) the placement of comment delimiters on blank lines. With this option enabled, comments look like this:

```
/*
 * this is a comment
 */
```

Rather than like this:

```
/* this is a comment */
```

This only affects block comments, not comments to the right of code. The

default is **-cdb**.

-ce,-nce Enables (disables) forcing 'else's to cuddle up to the immediately preceding '}'. The default is **-ce**.

-ci*n* Sets the continuation indent to be *n*. Continuation lines will be indented that far from the beginning of the first line of the statement. Parenthesized expressions have extra indentation added to indicate the nesting, unless **-lp** is in effect. **-ci** defaults to the same value as **-i**.

-cli*n* Causes case labels to be indented *n* tab stops to the right of the containing **switch** statement. **-cli0.5** causes case labels to be indented half a tab stop. The default is **-cli0**. (This is the only option that takes a fractional argument.)

-d*n* Controls the placement of comments which are not to the right of code. Specifying **-d1** means that such comments are placed one indentation level to the left of code. The default **-d0** lines up these comments with the code. See the section on comment indentation below.

-di*n* Specifies the indentation, in character positions, from a declaration keyword to the following identifier. The default is **-di16**.

-dj,-ndj **-dj** left justifies declarations. **-ndj** indents declarations the same as code. The default is **-ndj**.

-ei,-nei Enables (disables) special **else-if** processing. If enabled, **if**s following **else**s will have the same indentation as the preceding **if** statement. The default is **-ei**.

-fc1,-nfc1 Enables (disables) the formatting of comments that start in column 1. Often, comments whose leading '/' is in column 1 have been carefully hand formatted by the programmer. In such cases, **-nfc1** should be used. The default is **-fc1**.

-fcd,-nfcd Enables (disables) the formatting of compiler directive comments. Compiler directive comments are comments whose first character is non-white space, such as, */*\$*scalar**/* or */*VARARGS*/*. Compilers often require that there be no space between the '/*' and the directive. In such cases, **-nfcd** should be used. The default is **-fcd**.

-i*n* The number of spaces for one indentation level. The default is 8.

-ip,-nip Enables (disables) the indentation of parameter declarations from the left margin. The default is **-ip**.

-l*n* Maximum length of an output line. The default is 78.

-lp,-nlp Lines up code surrounded by parenthesis in continuation lines. If a line has a left paren which is not closed on that line, then continuation lines will be lined up to start at the character position just after the left paren. For example, here is how a piece of continued code looks with **-nlp** in effect:

```
p1 = first_procedure(second_procedure(p2, p3),
    third_procedure(p4, p5));
```

With **-lp** in effect (the default) the code looks somewhat clearer:

```
p1 = first_procedure(second_procedure(p2, p3),
                     third_procedure(p4, p5));
```

Inserting two more newlines we get:

```
p1 = first_procedure(second_procedure(p2,
                                      p3),
                     third_procedure(p4,
                                     p5));
```

-npro Causes the profile files, './.indent.pro' and '~/.indent.pro', to be ignored.

–pcs,–npcs If true (**–pcs**) all procedure calls will have a space inserted between the name and the '('. The default is **–npcs**.

–ps,–nps If true (**–ps**) the pointer following operator '–>' will be surrounded by spaces on either side. The default is **–nps**.

–psl,–npsl If true (**–psl**) the names of procedures being defined are placed in column 1 – their types, if any, will be left on the previous lines. The default is **–psl**.

–sc,–nsc Enables (disables) the placement of asterisks ('*'s) at the left edge of all comments. The default is **–sc**.

–sob,–nsob If **–sob** is specified, *indent* will swallow optional blank lines. You can use this to get rid of blank lines after declarations. Default: **–nsob**.

–st Causes *indent* to take its input from stdin, and put its output to stdout.

–T*typename* Adds *typename* to the list of type keywords. Names accumulate: **–T** can be specified more than once. You need to specify all the typenames that appear in your program that are defined by **typedef**s – nothing will be harmed if you miss a few, but the program won't be formatted as nicely as it should. This sounds like a painful thing to have to do, but it's really a symptom of a problem in C: **typedef** causes a syntactic change in the language and *indent* can't find all **typedef**s.

–troff Causes *indent* to format the program for processing by troff. It will produce a fancy listing in much the same spirit as **vgrind**. If the output file is not specified, the default is standard output, rather than formatting in place.

–v,–nv **–v** turns on 'verbose' mode; **–nv** turns it off. When in verbose mode, *indent* reports when it splits one line of input into two or more lines of output, and gives some size statistics at completion. The default is **–nv**.

FURTHER DESCRIPTION

You may set up your own 'profile' of defaults to *indent* by creating a file called *.indent.pro* in either your login directory and/or the current directory and including whatever switches you like. Switches in '.indent.pro' in the current directory override those in your login directory (with the exception of **-T** type definitions, which just accumulate). If *indent* is run and a profile file exists, then it is read to set up the program's defaults. The switches should be separated by spaces, tabs or newlines. Switches on the command line, however, override profile switches.

Comments

'Box' comments. *indent* assumes that any comment with a dash or star immediately after the start of comment (that is, '/*–' or '/**') is a comment surrounded by a box of stars. Each line of such a comment is left unchanged, except that its indentation may be adjusted to account for the change in indentation of the first line of the comment.

Straight text. All other comments are treated as straight text. *indent* fits as many words (separated by blanks, tabs, or newlines) on a line as possible. Blank lines break paragraphs.

Comment indentation

If a comment is on a line with code it is started in the 'comment column', which is set by the **–c***n* command line parameter. Otherwise, the comment is started at *n* indentation levels less than where code is currently being placed, where *n* is specified by the **–d***n* command line parameter. If the code on a line extends past the comment column, the comment starts further to the right, and the right margin may be automatically extended in extreme cases.

Preprocessor lines

In general, *indent* leaves preprocessor lines alone. The only reformatting that it will do is to straighten up trailing comments. It leaves embedded comments alone. Conditional compilation (**#ifdef...#endif**) is recognized and *indent* attempts to correctly compensate for the syntactic peculiarities introduced.

C syntax

indent understands a substantial amount about the syntax of C, but it has a 'forgiving' parser. It attempts to cope with the usual sorts of incomplete and misformed syntax. In particular, the use of macros like:

```
#define forever for(;;)
```

is handled properly.

FILES

./.indent.pro	profile file
~/.indent.pro	profile file

BUGS

indent has even more switches than *ls*.

A common mistake that often causes grief is typing:

```
indent *.c
```

to the shell in an attempt to indent all the C programs in a directory. This is probably a bug, not a feature.

NAME

info – menu driven online information system

SYNOPSIS

info [*command*] [*topic*]
info –audit
info –noaudit

DESCRIPTION

Info provides descriptions of ConvexOS commands with examples. It is meant to be easier to use and somewhat less confusing than the ConvexOS man pages. *info* also tells you where to look in CONVEX user manuals for information on ConvexOS commands. For each command, related ConvexOS commands are listed and described. Although *info* is an interactive system designed mainly for novices, the *Examples* section in particular may help any ConvexOS user.

When you invoke the Info System for the first time, you get an introduction which outlines the basic layout and features of the system. When you invoke *info* again, the system bypasses the introduction.

You can invoke *info* three ways. Without an argument, *info* displays the main menus. With a topic name as an argument, *info* displays the menu or command description relating to the specified topic. With a command name as an argument, *info* displays the command description screen for the specified command. If you list more than one command/topic argument, they are processed in the order they are listed. If you press the **ESC** key, *info* stops processing the arguments and displays the main menu.

There are three main types of screens in the Info System: the main menu, submenus, and command description screens. Each of these screen types has an associated help screen which explains use of that screen. As a reminder, each screen in the Info System includes a reverse video line at the bottom which lists the valid options executable from that screen.

If you are not sure of the exact command for which you need information, the menus are designed to lead you to an appropriate command description screen by narrowing down the chosen topic with each lower-level submenu you enter. Eventually, a selection you make from a submenu takes you to a command description screen rather than another submenu.

You can browse through a list of covered commands and related topics from the main menu (the first menu you will see when you invoke the Info System). Looking through this list may help you to pinpoint a keyword that describes the function you would like to perform. You can access the main menu from anywhere in the tree by pressing the **ESC** key, and you can quit the program at any time by pressing **q**. Options that exist on most screens include:

q (quit) exits the Info System.

ESC (escape) displays the main menu. This command does not work from the main menu.

n (next) moves forward one screen.

b (back) moves back one screen. If you are in a submenu, it takes you to the previous menu. If you are on the first screen of a command description, you go to the previous menu. This command does not work from the main menu.

r (report) allows the user to make a comment about *info* by invoking *contact* to create a bug report for the CONVEX Technical Assistance Center. This function only works if your site has a UUCP connection.

? (help) displays a context-sensitive help screen.

The standard backspace, delete, clear screen (<^l>), and erase-to-end-of-line keys on your terminal work in the Info System as well. In the main menu and submenus, the following options are allowed:

topic displays the menu or command description that relates to the specified topic. If a menu or command description for the named topic is not found, the user can then check for the existence of a man page for a similarly-named command.

(topic/command list)
displays a list of topics and ConvexOS commands which are indexed in the Info System.

command displays the command description for the named command. If a command description for the named command is not found, the user can then select the man page for the named command.

number displays the menu choice that corresponds to the number.

On the command description screens, the following characters are also valid:

x (execute) prompts you for a ConvexOS command to execute.

m (man page) displays the ConvexOS man page on that command from the last screen of the command description, after you have viewed the simplified information provided by the Info System.

On the help screens, you may also press:

i (introduction) displays the introductory information.

The information files are filtered through the *nroff* and *ul* processors before being displayed. You can add your own information files in the */usr/infosys/localscreens* directory. Please see the README file in that subdirectory for more information.

NOTES

System Managers:

The latter two commands outlined in the SYNOPSIS section (**info -audit** and **info -noaudit**) are only executable by a superuser. They control the audit function, which gathers summaries on the use and usefulness of the Info System. The *info* program tracks the menus and commands that users select, so that at CONVEX we can detect possible topics that are not covered thoroughly enough. *info* automatically forwards these summaries to CONVEX for investigation (and then removes them from your system) monthly.

The intent of the audit files is to provide our customers with a better product. However, if these files appear to be growing too large and occupying too much space, you can turn the audit function off.

Call *info* with the **–noaudit** option to send the current audit files to CONVEX and turn off the audit mechanism for future calls to *info*. To turn back on the audit mechanism, call *info* with the **–audit** option. The audit files are stored and sent only if your site has a UUCP connection.

SEE ALSO

ul(1), man(1), catinfo(8)

NAME

install – install binaries

SYNOPSIS

install [**–c**] [**–p**] [**–m** mode] [**–o** owner] [**–g** group] [**–s**] [**–v** version] 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 owner ID defaults to the owner ID of the user invoking *install*; the **–o** *owner* option may be used to specify a different owner.

destination group ID defaults to the group ID of the user invoking *install*; the **–g** *group* option may be used to specify a different group.

When copying, if the **–p** option is specified, then the modification time of the source file is preserved on the destination.

If the **–s** option is specified, the binary is stripped after being installed.

If the **–v** option is specified, the version number is set to *version*.

install refuses to move a file onto itself.

SEE ALSO

chgrp(1), chmod(1), cp(1), mv(1), strip(1), vers(1), chown(8), chall(8)

NAME

join – relational database operator

SYNOPSIS

join [options] file1 file2

DESCRIPTION

Join forms, on the standard output, a join of the two relations specified by the lines of *file1* and *file2*. If *file1* is '–', the standard input is used.

File1 and *file2* must be sorted in increasing ASCII collating sequence on the fields on which they are to be joined, normally the first in each line.

There is one line in the output for each pair of lines in *file1* and *file2* that have identical join fields. The output line normally consists of the common field, then the rest of the line from *file1*, then the rest of the line from *file2*.

Fields are normally separated by blank, tab or newline. In this case, multiple separators count as one, and leading separators are discarded.

These options are recognized:

–a*n* In addition to the normal output, produce a line for each unpairable line in file *n*, where *n* is 1 or 2.

–e *s* Replace empty output fields by string *s*.

–j*n m* Join on the *m*th field of file *n*. If *n* is missing, use the *m*th field in each file.

–o *list* Each output line comprises the fields specified in *list*, each element of which has the form *n.m*, where *n* is a file number and *m* is a field number.

–t*c* Use character *c* as a separator (tab character). Every appearance of *c* in a line is significant.

SEE ALSO

sort(1), comm(1), awk(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, look* and *awk*(1) are wildly incongruous.

NAME

kill – terminate a process with extreme prejudice

SYNOPSIS

kill [–sig] processid ...
kill –l

DESCRIPTION

kill sends the TERM (terminate, 15) signal to the specified processes. If a signal name or number preceded by '–' is given as first argument, that signal is sent instead of terminate (see *sigvec*(2)). The signal names are listed by 'kill –l', and are as given in */usr/include/signal.h*, stripped of the common SIG prefix.

The terminate signal will kill processes that do not catch the signal; 'kill –9 ...' is a sure kill, as the KILL (9) signal cannot be caught. By convention, if process number 0 is specified, all members in the process group (i.e. processes resulting from the current login) are signaled (but beware: this works only if you use *sh*(1); not if you use *csh*(1)). Negative process numbers also have special meanings; see *kill*(2) for details.

The killed processes must belong to the current user unless he is the superuser.

The process number of an asynchronous process started with '&' is reported by the shell. Process numbers can also be found by using *ps*(1). *kill* is a built-in to *csh*(1); it allows job specifiers of the form "%..." as arguments so process id's are not as often used as *kill* arguments. See *csh*(1) for details.

SEE ALSO

csh(1), ps(1), kill(2), sigvec(2)

BUGS

A replacement for "kill 0" for *csh*(1) users should be provided.

NAME

last – indicate last logins of users and teletypes

SYNOPSIS

last [–N] [–f wtmp_file] [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.

The –f option specifies an alternate wtmp-like file to be used in place of the default.

If *last* is interrupted, it indicates how far the search has progressed in *wtmp*. If interrupted with a quit signal (generated by a control-\) *last* indicates how far the search has progressed so far, and the search continues.

FILES

/usr/adm/wtmp	login data base
/usr/adm/shutdownlog	which records shutdowns and reasons for same

SEE ALSO

lastcomm(1), utmp(5), ac(8),
"Accounting" chapter in the *CONVEX System Manager's Guide.*

NAME

lastcomm – show last commands executed in reverse order

SYNOPSIS

lastcomm [–f acct_file] [command name] ... [user name] ... [terminal name] ...

DESCRIPTION

lastcomm gives information on previously executed commands. With no arguments, *lastcomm* prints information about all the commands recorded during the current accounting file's lifetime. If called with arguments, only accounting entries with a matching command name, user name, or terminal name are printed. So, for example,

lastcomm a.out root ttyd0

would produce a listing of all the executions of commands named *a.out* by user *root* on the terminal *ttyd0*.

For each process entry, the following are printed.

The command name under which the process was called.
Flags, as accumulated by the accounting facilities in the system.
The name of the user who ran the process.
The terminal from which the command was executed.
The amount of cpu time used by the process (in seconds).
The time the process exited.

The flags are encoded as follows: "S" indicates the command was executed by the superuser, "F" indicates the command ran after a fork, but without a following *exec*, "D" indicates the command terminated with the generation of a *core* file, "X" indicates the command was terminated with the signal SIGTERM, and "f" indicates that the command ran as a fixed-schedule job at least part of the time.

The –f option specifies an alternate accounting file to be used in place of the default.

SEE ALSO

last(1), mpa(1), sigvec(2), acct(5), core(5),
"Accounting" chapter in the *CONVEX System Manager's Guide*.

NAME

ld – link editor

SYNOPSIS

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, producing an object module which can be either executed or become the input for a further *ld* run. (In the latter case, the **–r** option must be given to preserve the relocation information.) The output of *ld* is left in *a.out*. This file is made executable only if no errors occurred during the load.

The argument routines are concatenated in the order specified. The entry point of the output defaults to the symbol **start**. If **start** is not defined, **_main** is used. If **_main** is undefined, the beginning of the first routine is used (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 loader can be made to scan libraries multiple times with the use of the **–m** flag.

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. If the dictionary is not present, *ld* will refuse to process the library. The **–F** flag may be used to override this behavior.

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. It is erroneous to define these symbols.

OPTIONS

NOTE: The ordering of the flags **–l**, **–t**, **–u**, **–y**, **–L**, and **–NL** on the command line is significant because they are interpreted when they are encountered as the command line is scanned on the first pass. The ordering of the flags **–s**, **–x**, and **–X** is extremely important. They **must** precede any object files on the command line. If they are encountered after any objects or libraries are processed, an error message will be printed and the load will be aborted. So, if you use them, put them first on the command line. The ordering of all the other flags is not important.

–A*alias*=*symbol*

Alias all occurrences of *alias* to *symbol*. This logically renames *alias* to *symbol*, causing all references of the symbol *alias* to resolve to the object pointed to by *symbol*.

–A*file* Read an alias list from *file*. The alias list is in the form *alias*==*symbol*, listed one to a line.

–d Force definition of common storage even if the **–r** flag is present.

–D*xxxx* Take the next argument as a hexadecimal number and pad the data segment with zero bytes to the indicated length. If the length indicated is not a multiple of 4096, the final size of the data segment will be rounded up to the nearest multiple of 4096.

–e*name*

The following argument is taken to be the name of the entry point of the loaded program; the default is **start**, **_main**, or the first module loaded as described above.

–E*mode*

The next argument is used to set one of the execution mode bits of the output file produced. This flag may be used multiple times to set multiple bits.

The valid modes are:

demand	sets the demand page bit
prepage	sets the prepaged bit
nonswap	sets the nonswapped bit
posix	sets the posix executable bit
noposix	clears the posix executable bit

Once an executable has been built, you can not change the posix bit by invoking ld with a different **-E** setting.

–f*format*
Specify the floating point format expected of all object modules. Valid format requests are **i** and **n**, for IEEE and native floating point format, respectively. All modules will be checked for compatibility with the expected mode, and the link will not succeed if any module is in conflict.

–F Force the creation of an executable file even if unresolved external references still remain at the end of the load. The file produced does not, however, have the executable mode bits set; that will be the responsibility of the user. Also, if any out-of-date libraries are encountered, *ld* will execute *ranlib*(1) on them to bring their table of contents up to date.

–H*xxxx* Leave a hole of size *xxxx* at the end of the text segment; *xxxx* is specified in hex.
NOTE: The actual size of the hole may be larger then the value specified. After adding the amount specified, *ld* rounds the segment size up to the nearest multiple of 4096.

–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'. If that does not exist, '/usr/local/lib/lib*x*.a'. A library is searched when its name is encountered, so the placement of a **–l** is significant.

–L*path* Add *path* to the search path for libraries included via the **–l** flag. The *path* will be added to the front of the search path list. This flag may be specified more than once.

–m Causes *ld* to perform multiple scans of the libraries processed if unresolved externals remain at the normal end of the load.

–M*option*
NOTE: no space is allowed between the flag and the option. Produce a map of the executable loaded. If no option is given, a primitive map consisting of a list of the object files included is produced. Three predefined maps are available:

S for a short map, **M** for a medium map, and **L** for a long map. You may create your own map by specifying just the sections you wish output. The map section option letters are:

h	Map header
l	library modules included
o	object modules included
g	global symbols sorted by address
s	global symbols sorted by name
n	local symbols for each object file by name

The short map is made up of sections **h, l,** and **o**. The medium map is made up of sections **h, l, o, g,** and **s**. The long map is made up of all the sections listed.

The option **P** may be specified with any of the predefined maps or when specifying individual sections. It will cause most of the formatting to be removed from the map so that it

is more easily processed by other programs. See the *CONVEX Loader User's Guide* for more details about load maps.

–NL Causes *ld* to not search the default paths for libraries specified by the **–l** flag. This flag, in conjunction with **–L** flag, allows the user to completely redefine the search path for libraries.

–o*name*
The *name* argument after **–o** is used as the name of the *ld* output file, instead of **a.out**.

–p This option performs the following: output o_entry in opthdr and preserve relocation information in the presence of **-r** option; allow warnings about undefined references in the presence of **-F** option; and to obey **-d** option.

–r Generate relocation information 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.

–R*ring* Relocates the resulting output file to the ring number specified in the *ring* argument. This switch is typically used during the system-generation procedure to rebuild or reboot the operating system residing in ring 0. Because of the protection mechanisms in operation, it is impossible to execute a program which specifies a ring number less than 4 while the system is running. The default is ring 4. See the *CONVEX Architecture Reference* for more information.

–s 'Strip' the output; that is, remove the symbol table and relocation information to save space (this impairs the usefulness of the debuggers). This information can also be removed by *strip*(1).

–t (trace) Print the name of each file as it is processed.

–T The next argument is a hexadecimal number which sets the text segment origin relative to the ring specified. The default origin is 1000 hexadecimal (4096 decimal). The value specified must be a multiple of 4096.

–u*name*
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*x.x.x.x*
The next argument is taken as a version number to set in the output file. This is equivalent to running *vers*(1) on the output file after it is created. The default version number of the output file is the largest version number found in any object/library module processed during the load.

–w Suppress all warning messages which would be produced by *ld*.

–x Do not preserve local (nonglobal) symbols in the output symbol table; only enter external symbols. This option saves some space in the output file.

–X Save local symbols except for those whose names begin with 'L' or '.L' or are of type ABS. This option is used by *cc*(1) to discard internally generated labels while retaining symbols local to routines.

–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 '_', as external C, FORTRAN, and Pascal variables begin with underscores.)

RESERVED FLAGS

Several flags which were accepted by *ld* prior to CONVEX UNIX V6.1 are now no longer

supported. They will be trapped by the loader as reserved flags and an error message will be produced reporting the use of a reserved flag. The following flags are now reserved and may not be used.

-i
-n
-z
-N
-S

ENVIRONMENT VARIABLE

The loader prepends the value of the environment variable *LDOPTIONS* (if it is set) to the command line so that options need not be specified every time *ld* is invoked.

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), sod(1), fc(1f)(optional product)
CONVEX Compiler Utilities User's Guide

NAME

learn – computer aided instruction about ConvexOS

SYNOPSIS

learn [–directory] [subject [lesson]]

DESCRIPTION

Learn gives Computer Aided Instruction courses and practice in the use of the ConvexOS operating system, the C Shell, and the Berkeley text editors. To get started simply type **learn**. The program will ask questions to find out what you want to do. Some questions may be bypassed by naming a *subject*, and more yet by naming a *lesson*. You may enter the *lesson* as a number that *learn* gave you in a previous session. If you do not know the lesson number, you may enter the *lesson* as a word, and *learn* will look for the first lesson containing it. If the *lesson* is '–', *learn* prompts for each lesson; this is useful for debugging.

The *subject*'s presently handled are

files
editor
vi
morefiles
macros
eqn
C

There are a few special commands. The command 'bye' terminates a *learn* session and 'where' tells you of your progress, with 'where m' telling you more. The command 'again' re-displays the text of the lesson and 'again *lesson*' lets you review *lesson*.

The *–directory* option allows one to exercise a script in a nonstandard place.

FILES

/usr/lib/learn	subtree for all dependent directories and files
/usr/tmp/pl*	playpen directories

SEE ALSO

csh(1), ex(1)

BUGS

The main strength of *learn*, that it asks the student to use the real ConvexOS, also makes possible baffling mistakes. It is helpful, especially for nonprogrammers, to have someone near at hand who can answer questions during the first sessions.

Occasionally lessons are incorrect, sometimes because the local version of a command operates in a non-standard way. Such lessons may be skipped with the 'skip' command, but it takes some sophistication to recognize the situation.

To find a *lesson* given as a word, *learn* does a simple *fgrep*(1) through the lessons. It is unclear whether this sort of subject indexing is better than none.

Spawning a new shell is required for each of many user and internal functions.

NAME

leave – remind you when you have to leave

SYNOPSIS

leave [hhmm]

DESCRIPTION

Leave waits until the specified time, then reminds you that you have to leave. You are reminded 5 minutes and 1 minute before the actual time, at the time, and every minute thereafter. When you log off, *leave* exits just before it would have printed the next message.

The time of day is in the form hhmm where hh is a time in hours (on a 12 or 24 hour clock). All times are converted to a 12 hour clock, and assumed to be in the next 12 hours.

If no argument is given, *leave* prompts with "When do you have to leave?". A reply of newline causes *leave* to exit, otherwise the reply is assumed to be a time. This form is suitable for inclusion in a *.login* or *.profile*.

Leave ignores interrupts, quits, and terminates. To get rid of it you should either log off or use "kill –9" giving its process id.

SEE ALSO

calendar(1)

AUTHOR

Mark Horton

NAME

less – file pager, alternative to more

SYNOPSIS

less [-[+]**aABcCdeEfiImMnqQuUsw**] [-**b***N*] [-**h***N*] [-**x***N*] [-[**z**]*N*]
[-**P**[**mM**=]*string*] [-[**lL**]*logfile*] [+*cmd*]
[-**t***tag*] [*filename*]...

DESCRIPTION

Less is a program similar to *more* (1), but which allows backwards movement in the file as well as forward movement. Also, *less* does not have to read the entire input file before starting, so with large input files it starts up faster than text editors like *vi* (1). *Less* uses termcap (or terminfo on some systems), so it can run on a variety of terminals. There is even limited support for hard-copy terminals. (On a hardcopy terminal, lines which should be printed at the top of the screen are prefixed with an up-arrow.)

Commands are based on *more, emacs* and *vi*. Commands may be preceded by a decimal number, called N in the descriptions below. The number is used by some commands, as indicated.

Note: *less* exits upon receiving a SIGTERM, SIGHUP or SIGQUIT signal.

COMMANDS

In the following descriptions, ^X means control-X. ESC stands for the ESCAPE key; for example ESC-v means the two character sequence "ESCAPE", then "v".

H Help: display a summary of these commands. If you forget all the other commands, remember this one.

SPACE or f or ^F or ^V
Scroll forward N lines, default one window (see option -z below). If N is more than the screen size, only the final screenful is displayed. Warning: some systems use ^V as a special literalization character.

b or ^B or ESC-v
Scroll backward N lines, default one window (see option -z below). If N is more than the screen size, only the final screenful is displayed.

RETURN or ^N or e or ^E or j or ^J
Scroll forward N lines, default 1. The entire N lines are displayed, even if N is more than the screen size.

y or ^Y or ^P or k or ^K
Scroll backward N lines, default 1. The entire N lines are displayed, even if N is more than the screen size. Warning: some systems use ^Y as a special job control character.

d or ^D
Scroll forward N lines, default one half of the screen size. If N is specified, it becomes the new default for subsequent d and u commands.

u or ^U
Scroll backward N lines, default one half of the screen size. If N is specified, it becomes the new default for subsequent d and u commands.

r or ^L Repaint the screen.

^R If the environment variable EDITOR is set to emacs then this is the same as *?*. Otherwise, it is the same as *r*.

R Repaint the screen, discarding any buffered input. Useful if the file is changing while it is being viewed.

g or < or ESC-<
Go to line N in the file, default 1 (beginning of file). (Warning: this may be slow if N is large.)

G or > or ESC->
Go to line N in the file, default the end of the file. (Warning: this may be slow if N is large, or if N is not specified and standard input, rather than a file, is being read.)

p or % Go to a position N percent into the file. N should be between 0 and 100. (This works if standard input is being read, but only if *less* has already read to the end of the file. It is always fast, but not always useful.)

m ^@ ESC-SPACE
Followed by any lowercase letter, marks the current position with that letter.

' (Single quote.) Followed by any lowercase letter, returns to the position which was previously marked with that letter. Followed by another single quote, returns to the position at which the last "large" movement command was executed. All marks are lost when a new file is examined.

^X^X Same as single quote.

/pattern ^]pattern
Search forward in the file for the N-th line containing the pattern. N defaults to 1. The pattern is a regular expression, as recognized by *ed*. The search starts at the second line displayed (but see the -a option, which changes this).

?pattern
Search backward in the file for the N-th line containing the pattern. The search starts at the line immediately before the top line displayed.

/!pattern
Like /, but the search is for the N-th line which does NOT contain the pattern.

?!pattern
Like ?, but the search is for the N-th line which does NOT contain the pattern.

n Repeat previous search, for N-th line containing the last pattern (or NOT containing the last pattern, if the previous search was /! or ?!).

E [filename]
Examine a new file. If the filename is missing, the "current" file (see the N and P commands below) from the list of files in the command line is re-examined. If the filename is a pound sign (#), the previously examined file is re-examined.

^X^V or :e
Same as E. Warning: some systems use ^V as a special literalization character.

}[function]
Go to function using tags. It also stores the current location so you can return easily (see { command.)

^}[function]
If the EDITOR environment variable is set to emacs, then this is the same as ^/ . Otherwise it is the same as }.

{ Return to location where the last } command was typed.

N or :n or ^Xn
Examine the next file (from the list of files given in the command line). If a number N is specified (not to be confused with the command N), the N-th next file is examined.

P or :p or ^Xp ^^
Examine the previous file. If a number N is specified, the N-th previous file is examined.

= or ^G
Prints some information about the file being viewed, including its name and the line number and byte offset of the bottom line being displayed. If possible, it also prints the length of the file and the percent of the file above the last displayed line.

– Followed by one of the command line option letters (see below), this will change the setting of that option and print a message describing the new setting. If the option letter has a numeric value (such as -b or -h), or a string value (such as -P or -t), a new value may be entered after the option letter.

_ (Underscore.) Followed by one of the command line option letters (see below), this will print a message describing the current setting of that option. The setting of the option is not changed.

+cmd Causes the specified cmd to be executed each time a new file is examined. For example, +G causes *less* to initially display each file starting at the end rather than the beginning.

V Prints the version number of *less* being run.

q or :q or ZZ
Exits *less*.

The following two commands may or may not be valid, depending on your particular installation.

v Invokes an editor to edit the current file being viewed. The editor is taken from the environment variable EDITOR, or defaults to "vi".

! shell-command ^_ shell-command ^X! shell-command
Invokes a shell to run the shell-command given. A percent sign in the command is replaced by the name of the current file. "!!" repeats the last shell command. "!" with no shell command simply invokes a shell. In all cases, the shell is taken from the environment variable SHELL, or defaults to "sh".

OPTIONS

Command line options are described below. Most options may be changed while *less* is running, via the "–" command.

Options are also taken from the environment variable "LESS". For example, to avoid typing "less -options ..." each time *less* is invoked, you might tell *csh:*

setenv LESS "-options"

or if you use *sh:*

LESS="-options"; export LESS

The environment variable is parsed before the command line, so command line options override the LESS environment variable. If an option appears in the LESS variable, it can be reset to its default on the command line by beginning the command line option with "-+".

A dollar sign ($) may be used to signal the end of an option string. This is important only for options like -P which take a following string.

-a Normally, forward searches start just after the top displayed line (that is, at the second displayed line). Thus, forward searches include the currently displayed screen. The -a option causes forward searches to start just after the bottom line displayed, thus skipping the currently displayed screen. Used to be the -t option.

-A The -A option causes searches to start at the second SCREEN line displayed, as opposed to the default which is to start at the second REAL line displayed. For example, suppose a long real line occupies the first three screen lines. The default search will start at the second real line (the fourth screen line), while the -A option will cause the search to start at the second screen line (in the midst of the first real line). (This option is rarely useful.)

-b The -b*n* option tells *less* to use a non-standard number of buffers. Buffers are 1K, and normally 10 buffers are used (except if data in coming from standard input; see the -B option). The number *n* specifies a different number of buffers to use.

-B Normally, when data is coming from standard input, buffers are allocated automatically as needed, to avoid loss of data. The -B option disables this feature, so that only the default number of buffers are used. If more data is read than will fit in the buffers, the oldest data is discarded.

-c Normally, *less* will repaint the screen by scrolling from the bottom of the screen. If the -c option is set, when *less* needs to change the entire display, it will paint from the top line down.

-C The -C option is like -c, but the screen is cleared before it is repainted.

-d Normally, *less* will complain if the terminal is dumb; that is, lacks some important capability, such as the ability to clear the screen or scroll backwards. The -d option suppresses this complaint (but does not otherwise change the behavior of the program on a dumb terminal).

-e Normally the only way to exit less is via the "q" command. The -e option tells *less* to automatically exit the second time it reaches end-of-file.

-E The -E flag causes *less* to exit the first time it reaches end-of-file.

-f Normally, *less* displays control characters with the two character sequence "^c". The -f option causes *less* not to do any special processing of control characters; they are sent to the terminal unchanged. This option is useful when viewing *nroff* output piped through *ul* which may generate some terminal escape sequences for underline or stand-out mode, etc.

-h Normally, *less* will scroll backwards when backwards movement is necessary. The -h option specifies a maximum number of lines to scroll backwards. If it is necessary to move backwards more than this many lines, the screen is repainted in a forward direction. (If the terminal does not have the ability to scroll backwards, -h0 is implied.)

-i The -i option causes searches to ignore case; that is, uppercase and lowercase are considered identical. Also, text which is overstruck or underlined can be searched for.

-I Normally, the interrupt signal (SIGINT, ^C) causes *less* to re-enter command mode. When -I is specified, SIGINT causes immediate exit like SIGQUIT and SIGTERM. This flag is for compatibility with *more* .

-l The -l option, followed immediately by a filename, will cause *less* to copy its input to the named file as it is being viewed. This applies only when the input file is a pipe, not an ordinary file. If the file already exists, *less* will ask for confirmation before overwriting it.

-L The -L option is like -l, but it will overwrite an existing file without asking for confirmation.

If no log file has been specified, the -l and -L options can be used from within *less* to specify a log file. Without a file name, they will simply report the name of the log file.

-m Normally, *less* prompts with a colon. The -m option causes *less* to prompt verbosely (like *more),* with the percent into the file.

-M The -M option causes *less* to prompt even more verbosely than *more.*

-n The -n flag suppresses line numbers. The default (to use line numbers) may cause *less* to run more slowly in some cases, especially with a very large input file. Suppressing line numbers with the -n flag will avoid this problem. Using line numbers means: the line number will be displayed in the verbose prompt and in the = command, and the v command will pass the current line number to the editor.

-P The -P option provides a way to tailor the three prompt styles to your own preference. You would normally put this option in your LESS environment variable, rather than type it in with each *less* command. Such an option must either be the last option in the LESS variable, or be terminated by a dollar sign. -P followed by a string changes the default (short) prompt to that string. -Pm changes the medium (-m) prompt to the string, and -PM changes the long (-M) prompt. Also, -P= changes the message printed by the = command to the given string. All prompt strings consist of a sequence of letters and special escape sequences. See the section on PROMPTS for more details.

-q Normally, if an attempt is made to scroll past the end of the file or before the beginning of the file, the terminal bell is rung to indicate this fact. The -q option tells *less* not to ring the bell at such times. If the terminal has a "visual bell", it is used instead.

-Q Even if -q is given, *less* will ring the bell on certain other errors, such as typing an invalid character. The -Q option tells *less* to be quiet all the time; that is, never ring the terminal bell. If the terminal has a "visual bell", it is used instead.

-s The -s option causes consecutive blank lines to be squeezed into a single blank line. This is useful when viewing *nroff* output.

-t The -t option, followed immediately by a TAG, will edit the file containing that tag. For this to work, there must be a file called "tags" in the current directory, which was previously built by the *ctags* (1) command. This option may also be specified from within *less* (using the – command) as a way of examining a new file. Note: The old -t command is now -a.

-u If the -u option is given, backspaces are treated as printable characters; that is, they are sent to the terminal when they appear in the input.

-U If the -U option is given, backspaces are printed as the two character sequence "^H".

If neither -u nor -U is given, backspaces which appear adjacent to an underscore character are treated specially: the underlined text is displayed using the terminal's hardware underlining capability. Also, backspaces which appear between two identical characters are treated specially: the overstruck text is printed using the terminal's hardware boldface capability. Other backspaces are deleted, along with the preceding character.

-w Normally, *less* uses a tilde character to represent lines past the end of the file. The -w option causes blank lines to be used instead.

-x The -x*n* option sets tab stops every *n* positions. The default for *n* is 8.

-[z] When given a backwards or forwards window command, *less* will by default scroll backwards or forwards one screenful of lines. The -z*n* option changes the default scrolling window size to *n* lines. Note that the "z" is optional for compatibility with *more*.

+ If a command line option begins with +, the remainder of that option is taken to be an initial command to *less*. For example, +G tells *less* to start at the end of the file rather than the beginning, and +/xyz tells it to start at the first occurrence of "xyz" in the file. As a special case, +<number> acts like +<number>g; that is, it starts the display at the specified line number (however, see the caveat under the "g" command above). If the option starts with ++, the initial command applies to every file being viewed, not just the first one. The + command described previously may also be used to set (or change) an initial command for every file.

KEY BINDINGS

You may define your own *less* commands by using the program *lesskey* (1) to create a file called ".less" in your home directory. This file specifies a set of command keys and an action associated with each key. See the *lesskey* manual page for more details.

PROMPTS

The -P option allows you to tailor the prompt to your preference. The string given to the -P option replaces the specified prompt string. Certain characters in the string are interpreted specially. The prompt mechanism is rather complicated to provide flexibility, but the ordinary user need not understand the details of constructing personalized prompt strings.

A percent sign followed by a single character is expanded according to what the following character is:

%bX Replaced by the byte offset into the current input file. The b is followed by a single character (shown as X above) which specifies the line whose byte offset is to be used. If the character is a "t", the byte offset of the top line in the display is used, an "m" means use the middle line, a "b" means use the bottom line, and a "B" means use the line just after the bottom line.

%f Replaced by the name of the current input file.

%i Replaced by the index of the current file in the list of input files.

%lX Replaced by the line number of a line in the input file. The line to be used is determined by the X, as with the %b option.

%m Replaced by the total number of input files.

%pX Replaced by the percent into the current input file. The line used is determined by the X as with the %b option.

%s Replaced by the size of the current input file.

%t Causes any trailing spaces to be removed. Usually used at the end of the string, but may appear anywhere.

%x Replaced by the name of the next input file in the list.

If any item is unknown (for example, the file size if input is a pipe), a question mark is printed instead.

The format of the prompt string can be changed depending on certain conditions. A question mark followed by a single character acts like an "IF": depending on the following character, a condition is evaluated. If the condition is true, any characters following the question mark and condition character, up to a period, are included in the prompt. If the condition is false, such characters are not included. A colon appearing between the question mark and the period can be used to establish an "ELSE": any characters between the colon and the period are included in the string if and only if the IF condition is false. Condition characters (which follow a question mark) may be:

?a True if any characters have been included in the prompt so far.

?bX True if the byte offset of the specified line is known.

?e True if at end-of-file.

?f True if there is an input filename (that is, if input is not a pipe).

?lX True if the line number of the specified line is known.

?m True if there is more than one input file.

?n True if this is the first prompt in a new input file.

?pX True if the percent into the current input file of the specified line is known.

?s True if the size of current input file is known.

?x True if there is a next input file (that is, if the current input file is not the last one).

Any characters other than the special ones (question mark, colon, period, percent, and backslash) become literally part of the prompt. Any of the special characters may be included in the prompt literally by preceding it with a backslash.

Some examples:

?f%f:Standard input.

This prompt prints the filename, if known; otherwise the string "Standard input".

?f%f .?ltLine %lt:?pt%pt:?btByte %bt:-...

This prompt would print the filename, if known. The filename is followed by the line number, if known, otherwise the percent if known, otherwise the byte offset if known. Otherwise, a dash is printed. Notice how each question mark has a matching period, and how the % after the %pt is included literally by escaping it with a backslash.

?n?f%f .?m(file %i of %m) ..?e(END) ?x- Next\: %x..%t

This prints the filename if this is the first prompt in a file, followed by the "file N of N" message if there is more than one input file. Then, if we are at end-of-file, the string "(END)" is printed followed by the name of the next file, if there is one. Finally, any trailing spaces are truncated. This is the default prompt. For reference, here are the defaults for the other two prompts (-m and -M respectively). Each is broken into two lines here for readability only.

```
?n?f%f .?m(file %i of %m) ..?e(END) ?x- Next\: %x.:
        ?pB%pB\%:byte %bB?s/%s...%t

?f%f .?n?m(file %i of %m) ..?ltline %lt :byte %bB?s/%s ..
        ?e(END) ?x- Next\: %x.:?pB%pB\%..%t
```

And here is the default message produced by the = command:

```
?f%f .?m(file %i of %m) .?ltline %lt .
      byte %bB?s/%s. ?e(END) :?pB%pB\%..%t
```

SEE ALSO

lesskey(1), emacs(1), man(1), more(1), vi(1)

BUGS

When using the -f flag, *less* can get confused if lines are actually longer than the width of the screen since it cannot know the actual line length when escape sequences occur in the line.

WARNINGS

The = command and prompts (unless changed by -P) report the line number of the line at the top of the screen, but the byte and percent of the line at the bottom of the screen.

NAME

lesskey – specify key bindings for less

SYNOPSIS

lesskey [-o output] [input]

DESCRIPTION

Lesskey is used to specify a set of key bindings to be used by *less*. The input file is a text file which describes the key bindings, and the output file is a binary file which is used by *less*. If no input file is specified, standard input is used. If no output file is specified, $HOME/.less is used.

The input file consists of lines of the form:

string <whitespace> action <newline>

Whitespace is any sequence of one or more spaces and/or tabs. The "string" is the command key(s) which invoke the action. The string may be a single command key, or a sequence of up to 15 keys. The "action" is the name of the less action, from the list below. The characters in the "string" may appear literally, or be prefixed by a caret to indicate a control key. A backslash may be used to cause the following character to be taken literally. Characters which must be preceded by backslash include caret, space, tab and the backslash itself. A backslash followed by one to three octal digits may be used to specify a character by its octal value. Blank lines and lines which start with a pound sign (#) are ignored.

As an example, the following input file describes the set of default command keys used by less:

k	back-line
y	back-line
^K	back-line
^Y	back-line
^P	back-line
b	back-screen
^B	back-screen
\33v	back-screen
u	back-scroll
^U	back-scroll
?	back-search
E	examine
^X^V	examine
+	first-cmd
e	forw-line
j	forw-line
^E	forw-line
^J	forw-line
^M	forw-line
^N	forw-line
f	forw-screen
^F	forw-screen
\40	forw-screen
^V	forw-screen
d	forw-scroll
^D	forw-scroll
/	forw-search
G	goto-end
>	goto-end
\33>	goto-end
g	goto-line
<	goto-line
\33<	goto-line
'	goto-mark
^X^X	goto-mark
H	help

N	next-file
%	percent
p	percent
P	prev-file
q	quit
ZZ	quit
^L	repaint
^R	repaint
r	repaint
R	flush-repaint
n	repeat-search
m	set-mark
!	shell
=	status
^G	status
-	toggle-option
_	display-option
V	version
v	visual

Commands specified by *lesskey* take precedence over the default commands. A default command key may be disabled by including it in the key file with the action "invalid".

SEE ALSO

less(1)

NAME

lex – generator of lexical analysis programs

SYNOPSIS

lex [**–tvfn**] [file] ...

DESCRIPTION

lex generates programs to be used in simple lexical analysis of text. The input *files* (standard input default) contain regular expressions to be searched for, and actions written in C to be executed when expressions are found.

A C source program, "lex.yy.c" is generated, to be compiled thus:

```
cc lex.yy.c –ll
```

This program, when run, copies unrecognized portions of the input to the output, and executes the associated C action for each regular expression that is recognized.

The options have the following meanings.

–t Place the result on the standard output instead of in file *lex.yy.c*.

–v Print a one-line summary of statistics of the generated analyzer.

–n Opposite of **–v**; **–n** is default.

–f "Faster" compilation: don't bother to pack the resulting tables; limited to small programs.

EXAMPLE

```
lex lexcommands
```

would draw *lex* instructions from the file *lexcommands*, and place the output in *lex.yy.c*

```
%%
[A–Z] putchar(yytext[0]+'a'–'A');
[ ]+$ {}
[ ]+  putchar(' ');
```

is an example of a *lex* program that would be put into a *lex* command file. This program converts upper case to lower, removes blanks at the end of lines, and replaces multiple blanks by single blanks.

BUGS

The input buffer used for pattern matching, *yytext*, is limited to 10000 characters. If a string is accumulated that is longer than 10000 characters before matching a pattern, *yylex* will behave as if the end of file had been reached. A warning message will be printed, and EOF will be returned.

If a pattern is to have no actions associated with it, this must be explicitly stated by placing a single semicolon (;) or an empty set of braces ({}) in the action column next to that pattern. (See the middle rule in the example above.)

SEE ALSO

yacc(1), sed(1)

"LEX – Lexical Analyzer Generator" in the *ConvexOS Tutorial Papers*

NAME

lint – a C program verifier

SYNOPSIS

lint [*option* ...] *files* ...

DESCRIPTION

lint attempts to detect features of the C program *files* that are likely to be bugs, or non-portable, or wasteful. It also checks the type usage of the program more strictly than the compilers. Among the things that are currently found are unreachable statements, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions which return values in some places and not in others, functions called with varying numbers of arguments, and functions whose values are not used.

By default, it is assumed that all the *files* are to be loaded together; they are checked for mutual compatibility. Function definitions for certain libraries are available to *lint*; these libraries are referred to by a conventional name, such as '–lc', in the style of *ld*(1). Arguments ending in *.ln* are also treated as library files.

COMPATIBILITY MODES

CONVEX *lint* operates in one of four compatibility modes.

-ext This mode provides an extended implementation of ANSI C and an ANSI C and POSIX compatible library system. This is the default.

-std This option causes *lint* to enforce syntax in accordance with the ANSI C language definition. Certain extensions, notably the *long long* type, are not available in this mode. External functions are checked against ANSI C and POSIX P1003.1 conforming libraries. Many extensions provided by the default library system are not available.

-str This option causes *lint* to enforce syntax in accordance with the ANSI C language definition and to attempt to detect language features that cause it not to conform to ANSI C. Only the library functions defined by ANSI C are available.

-pcc Forces language and library interpretation based on the the original Kernighan and Ritchie definition, the Common C Compiler, and traditional Unix systems. This mode is compatible with previous versions of lint shipped by CONVEX.

PREPROCESSOR CONTROL OPTIONS

lint predefines the symbol *__lint*. In the backward-compatible mode the symbol *lint* is also predefined.

-D

-I

-U These options of *cc*(1) are recognized with the same meanings they have for the C compiler.

DIAGNOSTIC CONTROL

–d *name*

–d *name=e*

–d *name=w*

The *-d* option provides low level control of diagnostic output. The form *-d name* suppresses the named message. The form *-d name=w* causes the named message to be reported as a warning. The form *-d name=e* causes the named message to be reported as an error.

The following *names* can be used to control specific messages. The CONVEX C documentation on C compiler error messages provides examples for each of these diagnostic options.

arg_ptr_qual	Detects actual function parameters that do not have the same type qualifiers as those declared in the function prototype.
arg_ptr_ref	Detects actual function parameters that are not the same pointer type as those declared in the function prototype.
assign_in_condition	Detects assignment expressions in locations where conditional expressions are expected, such as *for* statements and *if* statements.
char_cvt_truncates	Indicates that a *char* data type is the target of a cast of an integer type larger than a *char* data type.
class_ignored	Indicates that an explicit storage class is used to declare a *struct* tag, *union* tag, *enum* tag, or *enum* member.
const_condition	Detects constant expressions in locations where conditional expressions are expected such as *if* and *switch* expressions.
const_not_init	Detects uninitialized constant variables.
cvt_changes_sign	Detects when an unsigned integer is assigned to a signed integer of the same size.
cvt_to_unsigned	Detects when an unsigned integer is assigned to a signed integer of the same size.
division_by_zero	Detects compile time division by zero.
dollar_names	Detects identifiers embedded with the $ character.
escape_range_sequence	Detects when an escape sequence in an integer constant is greater than 0xff.
eval_order	Detects when an expression has an undefined evaluation order.
float_suffix	Detects when the floating-point suffixes, f, F, l, or L, are used. This option has no effect in the strict and conforming compatibility modes.
function_parameter	Reports when a function is used as a function parameter; only function pointers can be used as a function parameter.
function_pointer_cast	Looks for non-function pointers that are assigned to function pointers.
hidden_arg	Indicates that the parameter of a function is hidden by an identifier with the same name declared in the outermost block of that function. This diagnostic affects only the backward-compatible mode.
hidden_extern	Indicates that a declaration using *extern* inside a function definition causes an identifier not in the block to be obscured. This diagnostic affects only the backward-compatible mode.
hides_outer	Indicates that an identifier prevents access to an identifier of the same name in an enclosing block.
implicit_decl	Detects when an implicit declaration is used because a function has not been declared previously.

incomplete_record	Indicates that a *union* or a *struct* was declared without any members.
int_cvt_truncates	Indicates that a *long long* variable is converted to a variable of type *int*.
integer_overflow	Detects when an integer constant larger than 64 bits is used.
long_long_suffix	Detects when the integer literal suffixes LL or ll are used. These suffixes are permitted only in the extended and backward-compatible modes because the *long long* data type is not available in the strict and conforming compatibility modes.
misplaced_lint_directive	Checks for a *lint* directive used in the wrong context.
neg_shift	Checks for right operands of shift operators that are negative constants.
negative_to_uns	Indicates when a negative constant is assigned to an unsigned type.
no_arg_type	Checks for function arguments declared without a data type.
no_external_declaration	Detects when no declarations are accessible to other compilation units.
non_int_bit_field	Check for bit fields that have type other than *int* or *unsigned int*.
nothing_declared	Detects empty declarations such as *int;*.
pointer_alignment_efficiency	Checks for operations that can result in inefficient memory alignment.
pp_argcount	Indicates that the number of arguments in the actual argument list of a macro does not match the number of arguments in the formal argument list.
pp_argsended	Indicates that an actual argument list of a function-like macro does not have a terminating right parenthesis.
pp_badstr	Indicates incorrect use of #, the stringizing operator that is used in macro definitions.
pp_badtp	Indicates that the result of combining the left and right operands of the ## macro definition operator is illegal.
pp_extra	Indicates that a preprocessor directive has extra arguments. One common example is when the *#endif* directive is followed by text.
pp_idexpected	Indicates that a *#ifdef* or *#ifndef* does not have a legal identifier as an argument, or a formal argument of a function-like macro is not a legal identifier.
pp_line_range	Indicates the argument for the *#line* preprocessor directive does not have a value between 1 and 32767, inclusive.
pp_macro_arg	Indicates that two or more formal macro arguments have the same name.
pp_macro_redefinition	Indicates that the replacement list for a function-like macro is not the same as its original definition. The preprocessor uses the most recent replacement list.
pp_macro_redefinition_cmdl	Indicates that the replacement list for a function-like macro is not the same as its original definition on the command line. The

	preprocessor uses the most recent replacement list.
pp_malformed_directive	Indicates that the proper syntax for a preprocessor directive was not used.
pp_old_dir	Indicates that the comment style syntax for a compiler directive (pragma) was used.
pp_parse	Indicates that a `#if` directive has an invalid integer expression.
pp_undef	Indicates that the argument of the `#undef` directive is not permitted.
pp_undef_cmdl	Indicates that you attempted to undefined a macro that cannot be undefined on the command line.
pp_unrecognized_directive	Detects a preprocessor directive that the preprocessor does not recognize.
pp_unrecognized_pragma	Detects a pragma that the compiler does not recognize.
set_but_not_used	Detects variables that are assigned a value but are not used.
shift_too_large	Checks for constant right operands of shift operands that are too large.
short_cvt_truncates	Indicates that an integral type larger than a `short` integer is being assigned to a `short` integer.
skip_to_char	This diagnostic indicates that the compiler is skipping to a character to recover from an error.
skip_to_eof	This diagnostic indicates that the compiler must skip to the end of a file to recover from an error.
strict_syntax	Detects the lack of a semicolon after the last member in a `struct` or `union` declaration or the presence of a comma after the last enumeration constant in an `enum` declaration list.
uns_compare_neg	Indicates a comparison between a `short unsigned` integer and a negative constant.
uns_compare_zero	Indicates a comparison between an unsigned integer and zero.
unsigned_suffix	Detects use of the integer literal suffixes *U* or *u*. This option is not effective with the *=e* setting.
varargs_on_proto	Checks for usage of *lint* directive *VARARGS* on a function that has a function prototype. The ellipsis operator should be used instead.
varargs_too_large	Reports an error condition when the *lint VARARGS* argument is greater than the number of formal arguments in the function it precedes.
void_pointer_cast	Indicates that a pointer to `void` is involved in a cast operation.

CONTROL OPTIONS

The following options are used to control the types of checks performed by *lint.*

-a Report assignments of *long* values to *int* variables.

-b Report *break* statements that cannot be reached. (This is not the default because, unfortunately, most *lex* and many *yacc* outputs produce dozens of such comments.)

-c Complain about casts which have questionable portability.

-h Apply a number of heuristic tests to attempt to detect bugs, improve style, and reduce waste.

-n Do not check compatibility against the standard library.

-u Do not complain about functions and variables used and not defined, or defined and not used (this is suitable for running *lint* on a subset of files out of a larger program).

-v Suppress complaints about unused arguments in functions.

-x Report variables referred to by *extern* declarations, but never used.

-z Do not complain about structures that are never defined (e.g., using a structure pointer without knowing its contents.).

MISCELLANEOUS OPTIONS

–B*dir* Finds substitute lint (*cpp*, *lint1*, *lint2*, and *errmsgc*) in the directory *dir*. If *dir* is not specified, the standard backup version in the directory */usr/lib/oldcc* is used instead. For example, the command `lint -B/usr/new` would invoke */usr/new/lint1* instead of the default */usr/lib/cc/lint1*.

-C Used to create lint libraries. If *files* are the C sources of a library *congress*, the command

lint –C*congress files . . .*

would create a lint library *llib-lcongress.ln* which would be suitable for "linting" programs using the library *congress*.

–fi Translate floating-point constants to IEEE format and perform floating-point operations in IEEE mode. This option requires that the machine be equipped with IEEE floating-point hardware. If no floating-point format is specified, the sitc default is used. Arithmetic performed under this option does not conform to the IEEE standard.

–float sp_ops

–float dp_ops

–float sp_const

–float dp_const

These options control the floating-point system used by *lint* and are provided for compatibility with *cc*. *sp_ops* and *dp_ops* control the precision of the operations performed on float operands. When *sp_ops* is specified, 32 bit operations are performed. When *dp_ops* is specified, the float operands are converted to double and 64 bit operations are performed.

sp_const and *dp_const* control the representation of unsuffixed floating-point constants (which are normally represented in double precision). *sp_const* causes these constants to be represented in single precision. Some loss of accuracy may result. *dp_const* explicitly invokes the default.

sp_ops is the default in all but the backward-compatible mode.

dp_const is the default in all modes.

–fn Translate floating-point constants to native CONVEX format and perform floating-point operations in native mode. If no floating-point format is specified, the site default is used.

–sso Treat the result of the sizeof operator as a signed value. The default is unsigned int. This is only available in the backward-compatible mode.

–tl *time*
Sets the maximum CPU time limit on *lint* to *time* minutes. If the CPU time exceeds the allotted time, *lint* is aborted.

–vn Identifies *lint* version. Outputs the version numbers of *lint*, *cpp*, *lint1*, *lint2* and *errmsgc* to *stderr*.

SOURCE LEVEL CONTROL

Certain conventional comments in the C source will change the behavior of *lint*:

/*NOTREACHED*/
At appropriate points, stops comments about unreachable code.

/*VARARGS*n*/
Suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0. Should only be used with non-prototyped function definitions.

/*ARGSUSED*/
Turns on the –v option for the next function.

/*LINTLIBRARY*/
At the beginning of a file, shuts off complaints about unused functions in this file.

ENVIRONMENT VARIABLES

CONVEX *lint* prepends the value of the environment variable **LINTOPTIONS** (if it is set) to each command line so that options need not be specified every time *lint* is invoked. For example,

```
setenv LINTOPTIONS '-d pointer_alignment_efficiency'
```

causes *lint* to not report pointer casts which could result in inefficiently aligned pointers and

```
setenv LINTOPTIONS '-pcc'
```

causes *lint* operate in the backward-compatible mode.

The preprocessor *cpp* has a similar variable **CPPOPTIONS** which can be used to affect its behavior when it is invoked directly. However, **CPPOPTIONS** does not have any effect when *lint* invokes *cpp* (lint removes the variable before invoking *cpp*).

FILES

/usr/bin/lint	lint driver
/usr/lib/cc/errmsgc	error message file
/usr/lib/cc/lint[12]	programs
/usr/lib/lint/llib-lc.ln	declarations for standard functions
/usr/lib/lint/llib-lc	human readable version of above
llib-l*.ln	library created with –C

SEE ALSO

cc(1)
"Lint, a C Program Checker" in the *ConvexOS Tutorial Papers*

BUGS

There are some things you just **can't** get *lint* to shut up about.

exit(2) and other functions which do not return are not understood; this causes various lies.

The library definition for -std and -str modes is the same as that of the -ext mode (see COMPATIBILITY MODES section).

NAME

ln – make links

SYNOPSIS

ln [**–s**] name1 [name2]
ln name ... directory

DESCRIPTION

A link is a directory entry referring to a file; the same file (together with its size, all its protection information, etc.) may have several links to it. There are two kinds of links: hard links and symbolic links.

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

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

Given one or two arguments, *ln* creates a link to an existing file *name1*. If *name2* is given, the link has that name; *name2* may also be a directory in which to place the link; otherwise it is placed in the current directory. If only the directory is specified, the link will be made to the last component of *name1*.

Given more than two arguments, *ln* makes links to all the named files in the named directory. The links made will have the same name as the files being linked to.

SEE ALSO

rm(1), cp(1), mv(1), link(2), readlink(2), stat(2), symlink(2)

NAME

lock – reserve a terminal

SYNOPSIS

lock [**-p**] [**-number**]

DESCRIPTION

Lock requests a password from the user, reads it again for verification and then it will normally not relinquish the terminal until the password is repeated. There are three other conditions under which it will terminate: it accepts the password for root as an alternative to the one given by the user, it will timeout after some interval of time, and it may be killed by somebody with the appropriate permission. The default time limit is 15 minutes but it may be changed with the *-number* option where *number* is the time limit in minutes. The *-p* option has *lock* use the user's standard password instead of requesting another one.

NOTE

lock will not work if root does not have a password.

NAME

logger – make entries in the system log

SYNOPSIS

logger [**–t** tag] [**–p** pri] [**–i**] [**–f** file] [message ...]

ARGUMENTS

–t *tag*	Mark every line in the log with the specified *tag*. If *-t* is not given, every log message will be prefixed with the date, time, hostname, and username of the caller. If the user is not on a logged in tty, the username is omitted.
–p *pri*	Enter the message with the specified priority. The priority may be specified numerically or as a "facility.level" pair. For example, "–p local3.info" logs the message(s) as *inf*ormational level in the *local3* facility. The default is "user.notice."
–i	Log the process id of the logger process with each line.
–f *file*	Log the specified file.
message	The message to log; if not specified, the **–f** file or standard input is logged.

DESCRIPTION

Logger provides a program interface to the *syslog*(3) system log module.

A message can be given on the command line, which is logged immediately, or a file is read and each line is logged.

EXAMPLES

logger System rebooted

logger –p local0.notice –t HOSTIDM –f /dev/idmc

SEE ALSO

syslog(3), syslogd(8)

NAME

login – sign on

SYNOPSIS

login [*username*]

DESCRIPTION

The *login* command is used when a user initially signs on, or it may be used at any time to change from one user to another. The latter case is the one summarized above and described here. See the *ConvexOS Primer* for how to dial up initially.

If *login* is invoked without an argument, it asks for a user name, and, if appropriate, a password. Echoing is turned off (if possible) during the typing of the password, so it will not appear on the written record of the session.

login will prompt for a secondary password if a user is logging in from a dial-up port. Any terminal which is designated as a dialup terminal in */etc/ttys* is considered a dial-up port. The secondary password demanded is the one for the user *dialin* in */etc/passwd*.

After a successful login, accounting files are updated and the user is informed of the existence of mail, and the message-of-the-day is printed, as is the time he last logged in (unless he has a *.hushlogin* file in his home directory – this is mostly used to make life easier for non-human users, such as *uucp*). The message-of-the-day is only displayed to a user on the first login following modification of the */etc/motd* file.

login initializes the user and group ID's and the working directory, then executes a command interpreter (usually *csh*(1)), according to specifications found in a password file. Argument 0 of the command interpreter is **–csh**, or more generally the name of the command interpreter with a leading dash (–) prepended.

login also initializes the environment *environ*(7) with information specifying home directory, command interpreter, terminal type (if available), and user name.

If the user is subject to the rules of password aging, and the password has become outdated (the maximum period for which the password was valid has passed), *login* will force the user to supply a new password. The new password must meet the same requirements as if it had been changed with *passwd*(1); *login* will prevent the user from logging in until the password is changed.

If the file */etc/nologin* exists, *login* prints its contents on the user's terminal and exits. This is used by *shutdown*(8) to stop users logging in when the system is about to go down.

When a user fails to type a valid username and password or fails to type the correct dialin password from a dialup, the failure is logged with *syslog(3)* facility.

If the */etc/passwd* file is not readable by the user, *login* exits to prevent users from logging in.

login is recognized by *sh*(1) and *csh*(1), and executed directly (without forking). In this case, the user's *.logout* file is not executed.

FILES

/etc/utmp	accounting
/usr/adm/wtmp	accounting
/usr/spool/mail/*	mail
/etc/dumpmsg	message returned if the user does not have any access to /etc/passwd.
/etc/motd	message-of-the-day
/etc/passwd	password file
/etc/pwrestrict	password restrictions file
/etc/nologin	stops login's
.hushlogin	makes login quieter

SEE ALSO

mail(1), passwd(1), quota(1), syslog(3), ttys(5), passwd(5), pwrestrict(5), utmp(5) environ(7), init(8), getty(8), shutdown(8)

DIAGNOSTICS

"Login incorrect," if the name or the password is bad.
"No Shell", "cannot open password file", "no directory": consult a programming counselor.

login will allow remote users to login in if their username can be authenticated with *ruserok(3x)*. If the username cannot be authenticated and the local username does not have a password then the login will be refused.

BUGS

An undocumented option (**-r**) is used by the remote login server, *rlogind*(8C), to force *login* to enter into an initial connection protocol.

NAME

look – find lines in a sorted list

SYNOPSIS

look [**–df**] string [file]

DESCRIPTION

look consults a sorted *file* and prints all lines that begin with *string*. Because *look* uses binary search, it is necessary for the file to be sorted in *folded ascii* order. This can be done with *sort*(1).

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

SYNOPSIS

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

lpmv – move jobs from one line printer spooling queue to another queue

SYNOPSIS

lpmv [**–a**] [**–P***printer*] [*user...*] [*job#...*] *destination*

DESCRIPTION

lpmv moves one or more jobs from a printer's spool queue to another printer's spool queue. Because the spooling directory is protected from users, using *lpmv* is normally the only method by which a user may move a job.

lpmv without any options moves the currently active job on the default printer, if it is owned by the caller, to the destination printer and restart the job.

If the **–a** flag is specified, *lpmv* moves all jobs owned by a particular user. If the superuser employs this flag without an explicit user specified, the entire spool queue is moved to the destination printer. The owner is determined by the caller's uid and and host name on the machine where the *lpr* command was invoked.

Specifying a user's name, or list of user names, causes *lpmv* to attempt to move any jobs queued belonging to that user (or users). This form of invoking *lpmv* is useful only to the superuser.

A user may move an individual job by specifying its job number. This number may be obtained from the *lpq*(1) program, e.g.

```
% lpq

1st    ken    13  (standard input)        100 bytes
% lpmv 13 swip
```

lpmv announces the names of any files it moves and is silent if there are no jobs in the queue that match the request list.

lpmv kills off an active daemon, if necessary, before moving any spooling files. If a daemon is killed, a new one is automatically restarted upon completion of file movement.

The **–P** option may be used to specify the queue associated with a specific printer (otherwise the default printer or the value of the PRINTER variable in the environment is used).

FILES

/etc/printcap	printer characteristics file
/usr/spool/*	spooling directories
/usr/spool/*/lock	lock file used to obtain the pid of the current daemon and the job number of the currently active job

SEE ALSO

lpr(1), lpq(**1**), hosts(5), lpd(8)

DIAGNOSTICS

"Permission denied" if the user tries to remove files other than his or her own.

BUGS

Because there are race conditions possible in the update of the lock file, the currently active job may be incorrectly identified.

Because *lpmv* is included only on CONVEX systems, users may only move jobs between printers on CONVEX hosts.

NAME

lpq – spool queue examination program

SYNOPSIS

lpq [+[n]] [-f] [-l] [-Pprinter] [job # ...] [user ...]

DESCRIPTION

lpq examines the spooling area used by *lpd*(8) for printing files on the line printer, and reports the status of the specified jobs or all jobs associated with a user. *lpq* invoked without any arguments reports on any jobs currently in the queue. A **–P** flag may be used to specify a particular printer, otherwise the default line printer is used (or the value of the PRINTER variable in the environment). If queue redirection is in effect on the specified printer, *lpq* will examine both the original print queue and the queue specified by the redirection. If a + argument is supplied, *lpq* displays the spool queue until it empties. Supplying a number immediately after the + sign indicates that *lpq* should sleep *n* seconds in between scans of the queue. If **–f** is given, the queue is displayed forever. **-f** overrides the loop until empty specified by + but will still use the optional sleep period given in +[n]. All other arguments supplied are interpreted as user names or job numbers to filter out only those jobs of interest.

For each job submitted (i.e. invocation of *lpr*(1)), *lpq* reports the user's name, current rank in the queue, the names of files comprising the job, the job identifier (a number which may be supplied to *lprm*(1) for removing a specific job), and the total size in bytes. The **–l** option causes information about each of the files comprising the job to be printed. Normally, only as much information as will fit on one line is displayed. Job ordering is dependent on the algorithm used to scan the spooling directory and is supposed to be FIFO (First in First Out). File names comprising a job may be unavailable (when *lpr*(1) is used as a sink in a pipeline) in which case the file is indicated as "(standard input)".

If *lpq* warns that there is no daemon present (i.e. due to some malfunction), the *lpc*(8) command can be used to restart the printer daemon.

FILES

/etc/termcap	for manipulating the screen for repeated display
/etc/printcap	to determine printer characteristics
/usr/spool/*	the spooling directory, as determined from printcap
/usr/spool/*/cf*	control files specifying jobs
/usr/spool/*/lock	the lock file to obtain the currently active job

SEE ALSO

lpmv(1), lpr(1), lprm(1), lpc(8), lpd(8)

BUGS

Due to the dynamic nature of the information in the spooling directory *lpq* may report unreliably. Output formatting is sensitive to the line length of the terminal; this can result in widely spaced columns.

DIAGNOSTICS

Unable to open various files. The lock file being malformed. Garbage files when there is no daemon active, but files in the spooling directory.

NAME

lpr – off line print

SYNOPSIS

lpr [**–P***printer*] [**–#** *num*] [**–C** *class*] [**–J** *job*] [**–T** *title*] [**–i** [*numcols*]] [**–1234** *font*] [**–w***num*] [**–pltndgvcfrmhs**] [name ...]

DESCRIPTION

Lpr uses a spooling daemon to print the named files when facilities become available. If no names appear, the standard input is assumed. The **–P** option may be used to force output to a specific printer. Normally, the default printer is used (site dependent), or the value of the environment variable PRINTER is used. If queue redirection is in effect on the specified printer, *lpr* will submit the job to the queue specified by the redirection and notify the user of the redirection.

The following single letter options are used to notify the line printer spooler that the files are not standard text files. The spooling daemon will use the appropriate filters to print the data accordingly.

–p Use *pr*(1) to format the files (equivalent to *print*).

–l Use a filter which allows control characters to be printed and suppresses page breaks.

–t The files are assumed to contain data from *troff*(1) (cat phototypesetter commands).

–n The files are assumed to contain data from *ditroff* (device independent troff).

–d The files are assumed to contain data from *tex*(l) (DVI format from Stanford).

–g The files are assumed to contain standard plot data as produced by the *plot*(3X) routines (see also *plot*(1G) for the filters used by the printer spooler).

–v The files are assumed to contain a raster image for devices like the Benson Varian.

–c The files are assumed to contain data produced by *cifplot*(l).

–f Use a filter which interprets the first character of each line as a standard FORTRAN carriage control character.

The remaining single letter options have the following meaning.

–r Remove the file upon completion of spooling or upon completion of printing (with the **–s** option).

–m Send mail upon completion.

–h Suppress the printing of the burst page.

–s Use symbolic links. Usually files are copied to the spool directory.

The **–C** option takes the following argument as a job classification for use on the burst page. For example,

```
lpr –C EECS foo.c
```

causes the system name (the name returned by *hostname*(1)) to be replaced on the burst page by EECS, and the file foo.c to be printed.

The **–J** option takes the following argument as the job name to print on the burst page. Normally, the first file's name is used.

The **–T** option uses the next argument as the title used by *pr*(1) instead of the file name. The **-T** option is useful only when the **-p** option is also included.

To get multiple copies of output, use the **–#***num* option, where *num* is the number of copies desired of each file named. For example,

```
lpr –#3 foo.c bar.c more.c
```

would result in 3 copies of the file foo.c, followed by 3 copies of the file bar.c, etc. On the other hand,

```
cat foo.c bar.c more.c | lpr –#3
```

will give three copies of the concatenation of the files.

The **–i** option causes the output to be indented. If the next argument is numeric, it is used as the number of blanks to be printed before each line; otherwise, 8 characters are printed.

The **–w** option takes the immediately following number to be the page width for *pr*.

The **–s** option will use *symlink*(2) to link data files rather than trying to copy them so large files can be printed. This means the files should not be modified or removed until they have been printed.

The option **–1234** Specifies a font to be mounted on font position *i*. The daemon will construct a *.railmag* file referencing */usr/lib/vfont/name.size*.

FILES

/etc/passwd	personal identification
/etc/printcap	printer capabilities data base
/usr/lib/lpd*	line printer daemons
/usr/spool/*	directories used for spooling
/usr/spool/*/cf*	daemon control files
/usr/spool/*/df*	data files specified in "cf" files
/usr/spool/*/tf*	temporary copies of "cf" files

SEE ALSO

lpmv(1), lpq(1), lprm(1), pr(1), symlink(2), printcap(5), lpc(8), lpd(8)

DIAGNOSTICS

If you try to spool too large a file, it will be truncated. The maximum file size is set by the 'mx' entry in /etc/printcap. See *printcap*(5) for more information.

Lpr will object to printing binary files.

If a user other than root prints a file and spooling is disabled, *lpr* will print a message saying so and will not put jobs in the queue.

If a connection to *lpd* on the local machine cannot be made, *lpr* will say that the daemon cannot be started.

Diagnostics may be printed in the daemon's log file regarding missing spool files by *lpd*.

NAME

lprm – remove jobs from the line printer spooling queue

SYNOPSIS

lprm [**–P***printer*] [–] [job # ...] [user ...]

DESCRIPTION

lprm will remove a job, or jobs, from a printer's spool queue. Since the spooling directory is protected from users, using *lprm* is normally the only method by which a user may remove a job.

lprm without any arguments will delete the currently active job if it is owned by the user who invoked *lprm*.

If the – flag is specified, *lprm* will remove all jobs which a user owns. If the superuser employs this flag, the spool queue will be emptied entirely. The owner is determined by the user's login name and host name on the machine where the *lpr* command was invoked.

Specifying a user's name, or list of user names, will cause *lprm* to attempt to remove any jobs queued belonging to that user (or users). This form of invoking *lprm* is useful only to the superuser.

A user may dequeue an individual job by specifying its job number. This number may be obtained from the *lpq*(1) program, e.g.

```
% lpq –l

1st: ken                        [job #013ucbarpa]
    (standard input)            100 bytes
% lprm 13
```

lprm will announce the names of any files it removes and is silent if there are no jobs in the queue which match the request list.

lprm will kill off an active daemon, if necessary, before removing any spooling files. If a daemon is killed, a new one is automatically restarted upon completion of file removals.

The **–P** option may be used to specify the queue associated with a specific printer (otherwise the default printer, or the value of the PRINTER variable in the environment is used). If queue redirection is in effect, *lprm* will first scan the original queue, then the queue specified by the redirection, for the requested jobs.

FILES

/etc/printcap	printer characteristics file
/usr/spool/*	spooling directories
/usr/spool/*/lock	lock file used to obtain the pid of the current daemon and the job number of the currently active job

SEE ALSO

lpmv(1), lpq(1), lpr(1), hosts(5), lpd(8)

DIAGNOSTICS

"Permission denied" if the user tries to remove files other than his own.

NOTE

If the fields of */etc/hosts* are out of order, attempts to use *lprm* to remove jobs queued on remote machines will fail.

BUGS

Since there are race conditions possible in the update of the lock file, the currently active job may be incorrectly identified.

NAME

lprman, lprx – process nroff-style tty37 output for Printronix printers

SYNOPSIS

lprman [**–P***printer*] [**-l***n*] [**-h***n*] [**-v***n*] [**-t***n*] [*file* ...]
lprx [**–P***printer*] [**-l***n*] [**-h***n*] [**-v***n*] [**-t***n*] [*file* ...]

DESCRIPTION

lprman and *lprx* are filters which correctly format *nroff*-style tty37 output to print on Printronix printers. *lprx* prints 56 line pages with lines about 80 columns wide; *lprman* correctly prints manual pages generated via:

man name | lprman

The **-P** option may be used to force output to a specific printer. This option works as described in *lpr*(1).

The following options are used with both *lprman* and *lprx* to position text on the page:

-l*n* Set the page size to *n* lines.

-h*n* Set the horizontal margin to *n* columns.

-v*n* Set the position of the first line on the page to *n* lines from the top edge of the paper.

-t*n* Set tab size to *n* spaces. Default is 8 spaces.

-e Set 8 lines per inch and 74 lines per page. Nondefault.

The defaults are:

lprx: -h5 -v5 -l56
lprman: -h12 -v0 -l66

SEE ALSO

lpr(1)

NAME

ls, lf, ll, lr – list contents of directory

SYNOPSIS

ls [**–acdfghilqrstu1ACHLFR**] name ...
lf [**–acdfghilqrstu1ACHLR**] name ...
ll [**–acdfghiqrstu1ACHLFR**] name ...
lr [**–acdfghilqrstu1ACHLF**] name ...

DESCRIPTION

For each directory argument, *ls* lists the contents of the directory; for each file argument, *ls* repeats its name and any other information requested. There are three common options that have builtin abbreviations: *lf* is equivalent to *ls -F*, *ll* is interpreted as *ls -l*, and *lr* will display the same information as *ls -R*. Each of these special forms will also take any of the additional options listed below. By default, the output is sorted alphabetically. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments are processed before directories and their contents.

There are a large number of options:

–l List in long format, giving mode, number of links, owner, size in bytes, and time of last modification for each file. (See below.) If the file is a special file the size field will instead contain the major and minor device numbers. If the file is a symbolic link the pathname of the linked-to file is printed preceded by "–>".

–g Include the group ownership of the file in a long output.

–t Sort by time modified (latest first) instead of by name.

–a List all entries; in the absence of this option, entries whose names begin with a period (.) are *not* listed.

–s Display the number of full or partial kilobytes of physical disk space allocated for the file by the file system. The amount of space allocated is affected by the file system block and fragment sizes, and includes any indirect blocks which the file system may need to point to the data blocks of the file. Since the file system does not allocate physical disk space for "holes" in files, it is possible to have a sparse file with a logical size of many megabytes consume only a few kilobytes of physical disk space.

–d If argument is a directory, list only its name; often used with **–l** to get the status of a directory.

–L If argument is a symbolic link, list the file or directory the link references rather than the link itself.

–r Reverse the order of sort to get reverse alphabetic or oldest first as appropriate.

–u Use time of last access instead of last modification for sorting (with the **–t** option) and/or printing (with the **–l** option).

–c Use the last modification time of the inode for sorting or printing.

–i For each file, print the i-number in the first column of the report.

–f Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off **–l**, **–t**, **–s**, and **–r**, and turns on **–a**; the order is the order in which entries appear in the directory.

–A List all entries, including those whose names begin with a period (.), except for the special files "." and "..".

–F cause directories to be marked with a trailing "/", sockets with a trailing "=", symbolic links with a trailing "@", and executable files with a trailing "*". Regular files, including execuatables, partially or completely migrated will have a trailing "&".

–R recursively list subdirectories encountered.

–1 force one entry per line output format; this is the default when output is not to a terminal.

–C force multi-column output; this is the default when output is to a terminal.

-q force printing of non-graphic characters in file names as the character "?"; this is the default when output is to a terminal.

-H (h) Print the major (minor) device number for character and block special devices in hex. Only meaningful when used with the **-l** option.

The mode printed under the **-l** option contains 10 characters which are interpreted as follows: the first character is

d if the entry is a directory;
b if the entry is a block-type special file;
c if the entry is a character-type special file;
l if the entry is a symbolic link;
p if the entry is a fifo (a.k.a. "named pipe") special file;
s if the entry is a socket,
m if the entry is a partially or fully migrated plain file, or
- if the entry is a resident plain file.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to owner permissions; the next to permissions to others in the same user-group; and the last to all others. Within each set the three characters indicate permission respectively to read, to write, or to execute the file as a program. For a directory, "execute" permission is interpreted to mean permission to search the directory. The permissions are indicated as follows:

r if the file is readable;
w if the file is writable;
x if the file is executable;
- if the indicated permission is not granted.

The group-execute permission character is given as **s** if the file has the set-group-id bit set and is executable by group; likewise the user-execute permission character is given as **s** if the file has the set-user-id bit set and is user-executable. If either of these locations is given as **S**, then the corresponding executable bit is not set, but the set-group-id or set-user-id bit is.

The last character of the mode (normally "x" or "-") is **t** if the 1000 bit of the mode is on. See *sticky*(8) for the meaning of this mode.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks is printed.

FILES

/etc/passwd to get user ID's for "ls -l".
/etc/group to get group ID's for "ls -g".

NOTES

ls is aware that files larger than two gigabytes may exist, and responds appropriately. The file size fields have maintained the same default size, but will expand as required to print out larger file sizes.

BUGS

Newline and tab are considered printing characters in file names.

The output device is assumed to be 80 columns wide.

The option setting based on whether the output is a teletype is undesirable as "ls -s" is much different than "ls -s | lpr". On the other hand, not doing this setting would make old shell scripts which used *ls* almost certain losers.

NAME

m4 – macro processor

SYNOPSIS

m4 [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 arguments, or if an argument is –, the standard input is read. The processed text is written on the standard output.

Macro calls have the form:

name(arg1,arg2, . . . , argn)

The (must immediately follow the name of the macro. If a defined macro name is not followed by a (, it is deemed to have no arguments. Leading unquoted blanks, tabs, and newlines are ignored while collecting arguments. Potential macro names consist of alphabetic letters, digits, and underscore (_), where the first character is not a digit.

Left and right single quotes (` ´) are used to quote strings. The value of a quoted string is the string stripped of the quotes.

The pound sign character # begins a comment. *M4* will ignore all characters after the pound sign up to a newline. If a # is desired as an argument to a macro, it should be quoted as a string: '#'.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. 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 if this is done, the original meaning is lost. Their values are null unless otherwise stated.

define Installs the second argument 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.

undefine Removes the definition of the macro named in its argument.

ifdef If the first argument is defined, the value is the second argument; otherwise, it is the third. If there is no third argument, the value is null. The word *unix* is predefined on the ConvexOS version of *m4*.

changequote Changes quote characters to the first and second arguments. *Changequote* without arguments restores the original values (i.e., ` ').

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 there is no argument. Text may be undiverted into another diversion. Undiverting discards the diverted text.

divnum Returns the value of the current output stream.

dnl Reads and discards characters up to and including the next newline.

ifelse Has three or more arguments. If the first argument is the same string as the second, then the value is the third argument. If not, and if there are more than 4 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.

eval Evaluates its argument as an arithmetic expression, using 32-bit arithmetic. Operators include +, −, *, /, %, ^ (exponentiation); relationals; parentheses.

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 ConvexOS command given in the first argument. No value is returned.

maketemp Fills in a string of XXXXX in its argument with the current process ID.

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.

NAME

mail – send and receive mail

SYNOPSIS

mail [**–v**] [**–i**] [**–n**] [**–s** *subject*] [*user ...*]
mail [**–v**] [**–i**] [**–n**] **–f** [*name*]
mail [**–v**] [**–i**] [**–n**] **–u** *user*

INTRODUCTION

mail is an intelligent mail processing system, that has a command syntax reminiscent of *ed* with lines replaced by messages.

The **–v** flag puts mail into verbose mode; the details of delivery are displayed on the users terminal. The **–i** flag causes tty interrupt signals to be ignored. This is particularly useful when using *mail* on noisy phone lines. The **–n** flag inhibits the reading of */usr/lib/Mail.rc*.

Sending mail. To send a message to one or more people, *mail* can be invoked with arguments that are the names of people to whom the mail will be sent. You are then expected to type in your message, followed by an EOT (control–D) at the beginning of a line. A subject may be specified on the command line by using the **–s** flag. (Only the first argument after the **–s** flag is used as a subject; be careful to quote subjects containing spaces.) The section below, labeled *Replying to or originating mail,* describes some features of *mail* available to help you compose your letter.

Reading mail. In normal use, *mail* is given no arguments and checks your mail out of the post office, then prints out a one line header of each message there. The current message is initially the first message (numbered 1) and can be printed using the **print** command (which can be abbreviated **p**). You can move among the messages much as you move between lines in *ed*, with the commands "+" and "–" moving backwards and forwards, and simple numbers.

Disposing of mail. After examining a message you can **delete** (**d**) the message or **reply** (**r**) to it. Deletion causes the *mail* program to forget about the message. This is not irreversible; the message can be **undeleted** (**u**) by giving its number, or the *mail* session can be aborted by giving the **exit** (**x**) command. Deleted messages will, however, disappear never to be seen again.

Forwarding mail. If you have changed usernames or will be temporarily using a different username locally or at another site, you can have mail forwarded to the new account by using the **.forward** file in your home directory. By specifying the new username in **.forward,** all mail received by the old username will be forwarded to the new username. The format of the **.forward** file is one entry per line for each username or command the mail is to be forwarded to. Forwarding is actually the responsibility of the mail transport agent, in this case *sendmail*(8). See the *sendmail*(8) manual page for more information.

Specifying messages. Commands such as **print** and **delete** can be given a list of message numbers as arguments to apply to a number of messages at once. Thus "delete 1 2" deletes messages 1 and 2, while "delete 1–5" deletes messages 1 through 5. The special name "*" addresses all messages, and "$" addresses the last message; thus the command **top,** which prints the first few lines of a message, could be used in "top *" to print the first few lines of all messages.

Replying to or originating mail. You can use the **reply** command to set up a response to a message, sending it back to the person who it was from. Text you then type in, up to an end-of-file, defines the contents of the message. While you are composing a message, *mail* treats lines beginning with the character "~" specially. For instance, typing "~m" (alone on a line) places a copy of the current message into the response, right shifting it by a tabstop. Other escapes set up subject fields, add and delete recipients to the message, and allow you to escape to an editor to revise the message or to a shell to run some commands. (These options are given in the summary below.)

Ending a mail processing session. You can end a *mail* session with the **quit (q)** command. Messages that have been examined go to your *mbox* file unless they have been deleted (in which case they are discarded). Unexamined messages go back to the post office. The **–f** option causes *mail* to read in the contents of your *mbox* (or the specified file) for processing; when you **quit**, *mail* writes undeleted messages back to this file. The **–u** flag is a short way of doing "mail **–f** /usr/spool/mail/user."

Personal and systemwide distribution lists. It is also possible to create a personal distribution lists so that, for instance, you can send mail to "cohorts" and have it go to a group of people. Such lists can be defined by placing a line like

alias cohorts bill ozalp jkf mark kridle@ucbcory

in the file *.mailrc* in your home directory. The current list of such aliases can be displayed with the **alias (a)** command in *mail*. System-wide distribution lists can be created by editing */usr/lib/aliases*, (see *aliases*(5) and *sendmail*(8)); these are kept in a different syntax. In mail you send, personal aliases xkill word are be expanded in mail sent to others so that they will be able to **reply** to the recipients. System-wide aliases are not expanded when the mail is sent, but any reply returned to the machine will have the system-wide alias expanded as all mail goes through *sendmail*.

Network mail (ARPA, UUCP, Berknet), See *mailaddr(7)* for a description of network addresses.

mail has a number of options that can be set in the *.mailrc* file to alter its behavior; thus "set askcc" enables the "askcc" feature. (These options are summarized below.) If these options are given in the *.mailrc* file, any line preceded by a pound sign (**#**) and a space ( ) is treated as a comment.

SUMMARY

(Adapted from the *Mail Reference Manual*)

Each command is typed on a line by itself and may take arguments following the command word. The command need not be typed in its entirety – the first command that matches the typed prefix is used. For commands that take message lists as arguments, if no message list is given, then the next message forward which satisfies the command's requirements is used. If there are no messages forward of the current message, the search proceeds backwards, and if there are no good messages at all, *mail* types "No applicable messages" and aborts the command.

–	Goes to the previous message and prints it out. If given a numeric argument *n*, goes to the *n*-th previous message and prints it.
?	Prints a brief summary of commands.
!	Executes the ConvexOS shell command that follows.
Print	(**P**) Like **print** but also prints ignored header fields. See also **print** and **ignore**.
Reply	(**R**) Reply to originator. Does not reply to other recipients of the original message.
Type	(**T**) Identical to the **Print** command.
alias	(**a**) With no arguments, prints all currently-defined aliases. With one argument, prints that alias. With more than one argument, creates a new or changes an old alias.
alternates	(**alt**) The **alternates** command is useful if you have accounts on several machines. It can be used to inform *mail* that the listed addresses are really you. When you **reply** to messages, *mail* will not send a copy of the message to any of the addresses listed on the *alternates* list. If the **alternates** command is given with no argument, the current set of alternate names is displayed.
chdir	(**c**) Changes the user's working directory to that specified, if given. If no directory is given, then changes to the user's login directory.

copy (**co**) The **copy** command does the same thing that **save** does, except that it does not mark the messages it is used on for deletion when you quit.

delete (**d**) Takes a list of messages as argument and marks them all as deleted. Deleted messages are not saved in *mbox*, nor are the available for most other commands.

dp (also **dt**) Deletes the current message and prints the next message. If there is no next message, *mail* says "at EOF."

edit (**e**) Takes a list of messages and points the text editor at each one in turn. On return from the editor, the message is read back in.

exit (**ex** or **x**) Effects an immediate return to the Shell without modifying the user's system mailbox, *mbox* file, or edit file in **–f**.

file (**fi**) The same as **folder**.

folders List, the names of the folders in your folder directory.

folder (**fo**) The **folder** command switches to a new mail file or folder. With no arguments, it tells you which file you are currently reading. If you give it an argument, it writes out changes (such as deletions) you have made in the current file and reads in the new file. Some special conventions are recognized for the name: **#** means the previous file, **%** means your system mailbox, **%user** means user's system mailbox, **&** means your *~/mbox* file, and **+folder** means a file in your *folder* directory.

from (**f**) Takes a list of messages and prints their message headers.

headers (**h**) Lists the current range of headers, which is an 18-message group. If a "+" argument is given, then the next 18-message group is printed, and if a "–" argument is given, the previous 18-message group is printed.

help A synonym for ?

hold (**ho**, also **preserve**) Takes a message list and marks each message therein to be saved in the user's system mailbox instead of in *mbox*. Does not override the **delete** command.

ignore **N.B.:** *Ignore* has been superseded by *retain*.
Adds the list of header fields named to the *ignored list*. Header fields in the ignore list are not printed on your terminal when you print a message. This command is very handy for suppression of certain machine-generated header fields. The **Type** and **Print** commands can be used to print a message in its entirety, including ignored fields. If **ignore** is executed with no arguments, it lists the current set of ignored fields.

mail (**m**) Takes as argument login names and distribution group names and sends mail to those people.

mbox Indicates that a list of messages be sent to *mbox* in your home directory when you quit. This is the default action for messages if you do *not* have the *hold* option set.

next (**n** like + or CR) Goes to the next message in sequence and types it. With an argument list, types the next matching message.

preserve (**pre**) A synonym for **hold**.

print (**p**) Takes a message list and types out each message on the user's terminal.

quit (**q**) Terminates the session, saving all undeleted, unsaved messages in the user's *mbox* file in the login directory, preserving all messages marked with **hold** or **preserve** or never referenced in the system mailbox, and removing all other messages from the system mailbox. If new mail has arrived during the session, the message "You have new mail" is given. If given while editing a mailbox file with the **–f**

flag, then the edit file is rewritten. A return to the Shell is effected, unless the rewrite of edit file fails, in which case the user can escape with the **exit** command.

reply (**r**) Takes a message list and sends mail to the sender and all recipients of the specified message. The default message must not be deleted.

respond A synonym for **reply**.

retain Adds the list of header fields named to the *retained list*. Only the header fields in the retain list are shown on your terminal when you print a message. All other header fields are suppressed. The **Type** and **Print** commands can be used to print a message in its entirety. If **retain** is executed with no arguments, it lists the current set of retained fields.

save (**s**) Takes a message list and a filename and appends each message in turn to the end of the file. The filename in quotes, followed by the line count and character count is echoed on the user's terminal.

set (**se**) With no arguments, prints all variable values. Otherwise, sets option. Arguments are of the form "option=value" or "option."

shell (**sh**) Invokes an interactive version of the shell.

size Takes a message list and prints out the size in characters of each message.

source (**so**) The **source** command reads *mail* commands from a file.

top Takes a message list and prints the top few lines of each. The number of lines printed is controlled by the variable **toplines** and defaults to five.

type (**t**) A synonym for **print**.

unalias Takes a list of names defined by **alias** commands and discards the remembered groups of users. The group names no longer have any significance.

undelete (**u**) Takes a message list and marks each message as *not* being deleted.

unread (**U**) Takes a message list and marks each message as *not* having been read.

unset Takes a list of option names and discards their remembered values; the inverse of **set**.

visual (**v**) Takes a message list and invokes the display editor on each message.

write (**w**) Similar to **save**, except that *only* the message body (*without* the header) is saved. Extremely useful for such tasks as sending and receiving source program text over the message system.

xit (**x**) A synonym for **exit**.

z *mail* presents message headers in windowfuls as described under the **headers** command. You can move *mail*'s attention forward to the next window with the **z** command. Also, you can move to the previous window by using **z–**.

Here is a summary of the tilde escapes, which are used when composing messages to perform special functions. Tilde escapes are only recognized at the beginning of lines. The name "tilde escape" is somewhat of a misnomer since the actual escape character can be set by the option **escape**.

~!command Execute the indicated shell command, then return to the message.

~b name ... Add the given names to the list of carbon copy recipients but do not make the names visible in the Cc: line ("blind" carbon copy).

~c name ... Add the given names to the list of carbon copy recipients.

~d Read the file "dead.letter" from your home directory into the message.

~e Invoke the text editor on the message collected so far. After the editing session is finished, you may continue appending text to the message.

~f messages Read the named messages into the message being sent. If no messages are specified, read in the current message.

~h Edit the message header fields by typing each one in turn and allowing the user to append text to the end or modify the field by using the current terminal erase and kill characters.

~m messages Read the named messages into the message being sent, shifted right one tab. If no messages are specified, read the current message.

~p Print the message collected so far, prefaced by the message header fields.

~q Abort the message being sent, copying the message to "dead.letter" in your home directory if **save** is set.

~r filename Read the named file into the message.

~s string Cause the named string to become the current subject field.

~t name ... Add the given names to the direct recipient list.

~v Invoke an alternate editor (defined by the VISUAL option) on the message collected so far. Usually, the alternate editor will be a screen editor. After you quit the editor, you may resume appending text to the end of your message.

~w filename Write the message onto the named file.

~|command Pipe the message through the command as a filter. If the command gives no output or terminates abnormally, retain the original text of the message. The command *fmt*(1) is often used as *command* to rejustify the message.

~~string Insert the string of text in the message prefaced by a single ~. If you have changed the escape character, then you should double that character in order to send it.

Options are controlled via the **set** and **unset** commands. Options may be either binary, in which case it is only significant to see whether they are set or not; or string, in which case the actual value is of interest. The binary options include the following:

append Causes messages saved in *mbox* to be appended to the end rather than prepended. (This is set in */usr/lib/Mail.rc* on version 7 systems.)

ask Causes *mail* to prompt you for the subject of each message you send. If you respond with simply a newline, no subject field is sent.

askcc Causes you to be prompted for additional carbon copy recipients at the end of each message. Responding with a newline indicates your satisfaction with the current list.

autoprint Causes the **delete** command to behave like **dp:** after deleting a message, the next one is typed automatically.

debug Setting the binary option *debug* is the same as specifying **–d** on the command line and causes *mail* to output all sorts of information useful for debugging *mail*.

dot The binary option *dot* causes *mail* to interpret a period alone on a line as the terminator of a message you are sending.

hold This option is used to hold messages in the system mailbox by default.

ignore Causes interrupt signals from your terminal to be ignored and echoed as **@**'s.

ignoreeof An option related to *dot* is *ignoreeof* which makes *mail* refuse to accept a <CTRL-D> as the end of a message. *ignoreeof* also applies to *mail* command mode.

metoo Usually, when a group is expanded that contains the sender, the sender is removed from the expansion. Setting this option causes the sender to be included in the group.

nosave Normally, when you abort a message with two RUBOUT, *mail* copies the partial letter to the file "dead.letter" in your home directory. Setting the binary option *nosave* prevents this.

Replyall Reverses the sense of *reply* and *Reply* commands.

quiet Suppresses the printing of the version when first invoked.

verbose Setting the option *verbose* is the same as using the **–v** flag on the command line. When *mail* runs in verbose mode, the actual delivery of messages is displayed on the user's terminal.

The following options have string values:

EDITOR Pathname of the text editor to use in the **edit** command and **~e** escape. If not defined, then a default editor is used.

SHELL Pathname of the shell to use in the **!** command and the **~!** escape. A default shell is used if this option is not defined.

PAGER Pathname of the paging program to use when the number of lines in a message exceeds the number defined in **crt**. The default paging program is */usr/ucb/more*.

VISUAL Pathname of the text editor to use in the **visual** command and **~v** escape.

crt The valued option *crt* is used as a threshold to determine how long a message must be before *more* is used to read it. If you set *crt* to zero, your current window size is used.

escape If defined, the first character of this option gives the character to use in the place of **~** to denote escapes.

folder The name of the directory to use for storing folders of messages. If this name begins with a "/", *mail* considers it to be an absolute pathname; otherwise, the folder directory is found relative to your home directory.

record If defined, gives the pathname of the file used to record all outgoing mail. If not defined, then outgoing mail is not saved.

toplines If defined, gives the number of lines of a message to be printed out with the **top** command; normally, the first five lines are printed.

FILES

/usr/spool/mail/*	post office
~/mbox	your old mail
~/.mailrc	file giving initial mail commands
~/.forward	file of addresses to forward the mail to.
/tmp/R#	temporary for editor escape
/usr/lib/Mail.help*	help files
/usr/lib/Mail.rc	system initialization file
Message*	temporary for editing messages

SEE ALSO

binmail(1), fmt(1), newaliases(1), aliases(5),
mailaddr(7), sendmail(8)
"The Mail Reference Manual" in the *ConvexOS Tutorial Papers*

RESTRICTIONS

If two or more *Mail* processes try to modify the same file or folder, that file or folder could be corrupted. *Mail* tries to prevent this by allowing the first process read-write access, and giving the second (and Nth) process read-only access. However, if a folder is NFS mounted, this level of protection may not help.

BUGS

There are many flags that are not documented here. Most are not useful to the general user. Usually, *mail* is just a link to *Mail*, which can be confusing.

The *alternates* command may not work in complicated pathnames when using the *reply* option.

NAME

make – maintain program groups

SYNOPSIS

make [**–diknqrstvRSTW**] ... [**–j** *procnum*] [**–f** *alternatemakefile*] [**–b** *target*] *target* ...

DESCRIPTION

make executes a pre-defined set of commands to update one or more *targets*. *target* is typically a program. The commands that *make* executes are set up in a file that is usually named "makefile". You can use the **-f** option to specify a differently named *makefile*. If no **–f** option is present, *make* uses the set of commands in "makefile" or "Makefile", in that order. You can specify "-" as the *alternatemakefile* to enter input from your terminal.

make updates a *target* if any of its prerequisite files (listed in the *makefile*) have been modified since the *target* was last modified or if the *target* does not exist.

makefile contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated list of targets, then a colon, then a list of prerequisite files. Text following a semicolon and all following lines that begin with a tab are shell commands to be executed to update the target. If a name appears on the left of more than one "colon" line, then it depends on all of the names on the right of the colon on those lines, but only one command sequence may be specified for it. If a name appears on a line with a double colon (::), then the command sequence following that line is performed only if the name has a time of modification that is older than or equal to the names to the right of the double colon and is not affected by other double-colon lines on which that name may appear.

Two special forms of a name are recognized. A name like *ar.a*(*b.o*) means the file named *b.o* stored in the archive named *ar.a* . A name like *ar.a*((*b*)) means the file stored in archive *ar.a* containing the entry point *b*. When using the latter form, be aware of any language-specific external naming conventions, such as the fact that external identifiers generated by the C language begin with a leading underscore; for a C function named "b", the above example would more appropriately be written *ar.a*((*_b*)).

Comments begin on a separate line and are surrounded by the sharp (#) and newline symbols.

make also allows for the inclusion of one *makefile* into another *makefile*. Wherever the word "include" appears at the beginning of a line in the *makefile*, the word following it is used as the name of the file to be included. Any additional text on the line after the file name is ignored. The contents of the included file are then inserted at the point where the word "include" occurs. If the word "include" is preceded by a "-" then *make* will print a warning and continue execution if the included file is not found. If the word "include" is not preceded by a "-" and make does not find the included file then make will print an error message and exit with an error code of 1. If the word "include" is preceded by a "-", a warning message will be produced (unless –W was specified on the command line) and *make* will continue. Includes may be nested; i.e. a *makefile* that is included by another may itself include a third *makefile*.

The following makefile says that *pgm* depends on two files *a.o* and *b.o*, and that they in turn depend on *.c* files and a common file *incl*.

```
pgm: a.o b.o
        cc a.o b.o -lm -o pgm
a.o: incl a.c
        cc -c a.c
b.o: incl b.c
        cc -c b.c
```

makefile entries of the form:

string1 = string2

are macro definitions. Subsequent appearances of $(*string1*) or ${*string1*} are replaced by *string2*. If *string1* is a single character, the parentheses or braces are optional.

Macro references of the form

$(string:pat=rep)

perform substitution; all nonoverlapping occurrences of *pat* in the named macro are replaced by *rep*. For example,

```
OBJS=x1.o x2.o x3.o
SRCS=$(OBJS:.o=.c)
```

assigns to SRCS as if the *makefile* had contained

```
OBJS=x1.o x2.o x3.o
SRCS=x1.c x2.c x3.c
```

Macro references of the form

$(string:Lname)

perform an alternate substitution designed to construct lists of archive members. Each word in the macro named by *string* is prefixed by the value of the macro introduced by *L* and then parenthisized; note the requirement that the name of the macro containing the library name begin with an *L*. For example,

```
LMYLIB=mylib.a
OBJS=x1.o x2.o x3.o
LOBJS=$(OBJS:LMYLIB)
```

assigns to LOBJS as if the *makefile* had contained

```
OBJS=x1.o x2.o x3.o
LOBJS=mylib.a(x1.o) mylib.a(x2.o) mylib.a(x3.o)
```

make infers prerequisites for files for which the *makefile* gives no construction commands. For example, a *.c* file may be inferred as prerequisite for a *.o* file and be compiled to produce the *.o* file. Thus the preceding example can be done more briefly:

```
pgm: a.o b.o
        cc a.o b.o -lm -o pgm
a.o b.o: incl
```

Prerequisites are inferred according to selected suffixes listed as the "prerequisites" for the special name .SUFFIXES. Multiple lists accumulate; an empty list clears what came before. Order is significant; the first possible name for which both a file and a rule (as described in the next paragraph) exist is inferred. The default list is:

.SUFFIXES: .out .a .o .b .c .f .e .r .y .yr .ye .l .s .a68 .cl .p .h .c,v .f,v .e,v .r,v .y,v .yr,v .ye,v .l,v .s,v .a68,v .cl,v .p,v .h,v

The rule to create a file with suffix *s2* that depends on a similarly named file with suffix *s1* is specified as an entry for the "target" *.s1.s2*; a rule naming only *s1*, called a *single suffix rule*, allows building *name* from *name.s1P* . *In such entries, the special macro* $* stands for the target name with suffix deleted, $@ for the full target name, and $? for the list of prerequisites that are out of date. If the command was generated by an implicit rule, $< is the pathname of the related file that caused the action. $< is different from $* in that if the dependent file is in a

different directory, $< contains the pathname and the suffix. For example, a rule for making an optimized *.o* file from a *.c* file is:

```
.c.o: ; cc -c -O $*.c
```

For example, if the VPATH macro is set to *./foo* and the file *a.c* is in the subdirectory *./foo* then the $< macro must be used:

```
.c.o: ; cc -c -O $<
```

The macro $@ would expand to *a.o*, $* would be *a*, and $< would be *./foo/a.c*.

Additionally, the macro $% is evaluated when the target is an archive member of the form *ar.a(file.o)*. In this case, the macro $@ would expand to *ar.a* and $% would be *file.o*. This construct may be used as in the following example:

```
LIB = ar.a
LOBJ = $(LIB)(x1.o) $(LIB)(x2.o)

prog: main.o $(LIB)
        $(CC) -o $@ main.o $(LIB)

$(LIB): $(LOBJ)
        ranlib $@

.c.a:
        $(CC) -c $(CFLAGS) $*.c
        ar rv $@ $%
        rm -f $%
```

The macros $* , $@ , $< , and $% may appear with **D** or **F** appended, meaning the "directory" or "file" part of the macro, respectively. For instance, the following rules:

```
../make/lib.a: ../make/lib.a(main.o)
        echo $* $(*F) $(*D)
        echo $@ $(@F) $(@D)
        echo $< $(<F) $(<D)
```

would produce this output:

```
../make/lib lib ../make
../make/lib.a lib.a ../make
main.c main.c .
```

Certain macros are used by the default inference rules to communicate optional arguments to any resulting compilations. In particular, CFLAGS is used for *cc*(1) options, FFLAGS for *f77*(1) options, PFLAGS for *pc*(1) options, and LFLAGS and YFLAGS for *lex* and *yacc*(1) options. In addition, the macro MFLAGS is filled in with the initial command line options supplied to *make*. This simplifies maintaining a hierarchy of makefiles as one may then invoke *make* on makefiles in subdirectories and pass along useful options such as **–k**.

Two macros have special meaning only when concurrency is enabled with the **–j** option or by the existence of the environment variable MAKELIM. The SERIAL and ALONE macros should be set to a list of targets that are separated by spaces. All targets that are in the SERIAL list and their dependents will be built sequentially. The ALONE macro works similarly, except that *make* will wait until all other targets that are in the process of being built finish before starting a target that is in the ALONE list. If an implicit rule is included in the SERIAL or ALONE macro then all targets that are built by that implicit rule will be treated as if they had been included in the SERIAL or ALONE macro.

Another macro with special meaning is VPATH. The VPATH macro should be set to a list of directories separated by colons. When *make* searches for a file as a result of a dependency relation, it first searches the current directory and then each of the directories on the VPATH list. If the file is found, the actual path to the file is used, rather than just the filename. If VPATH is not defined, only the current directory is searched.

One use for VPATH is when one has several programs that compile from the same source. The source can be kept in one directory and each set of object files (along with a separate *makefile*) would be in a separate subdirectory. The VPATH macro would point to the source directory in this case. Please note that the VPATH macro only influences the behavior of make in implicit or explicit inference rules and no macros are allowed in the declaration of VPATH.

Command lines are executed one at a time, each by its own shell. If the environment variable MAKESHELL is set, it is names the command interpreter to be used; otherwise, /bin/sh is used. The shell is invoked with a first argument of *-ce* (or *-c* if the command began with a dash); the second argument is the command to execute.

Commands returning nonzero status (see *intro*(1)) cause *make* to terminate unless the special target .IGNORE is in *makefile* or the command begins with <tab><hyphen>.

<Interrupt> and <quit> cause the target to be deleted unless the target is a directory or depends on the special name .PRECIOUS.

The special target .DEFAULT, if present, is used when *make* determines the need to update a target for which there is neither an explicit nor a built-in rule.

Any filenames that end with *,v* are assumed to be Revision Control System (RCS) files. RCS files may have their modification times changed without updating the contents of the latest RCS revision (e.g. when locking a revision). The special target .RCSCHECK can be included in a *makefile* and forces *make* to examine the contents of the RCS file to use the creation time of the last RCS revision for the purpose of generating dependencies, rather than the modification time maintained by the ConvexOS file system. Specifying .RCSCHECK increases the amount of time required to analyze dependencies, but often decreases the likelihood that source files maintained by RCS will be recompiled unnecessarily.

make has the ability to build several targets concurrently. This allows for more efficient usage of multiprocessor machines. *make* searches for the environment variable MAKELIM. If MAKELIM exists and has a value greater than one, (valid values for MAKELIM are integer numbers) *make* will build targets in parallel. The value of MAKELIM is the maximum number of targets that can be built concurrently. The **–j** option can also be used to allow *make* to run in parallel. If both MAKELIM exists and the **–j** flag is given the value specified by the **–j** flag will override the value of the environment variable. If the values of the MAKELIM environment variable or **–j** exceed the built-in maximum values in *make*, the built-in maximums will used. Other command line options:

–b Ignore dependencies on the name passed as the next argument.

–d Debugging output. Displays file dependencies, actions, and rules used.

–i Equivalent to the special entry .IGNORE:.

–j If *procnum* is greater than one, *make* will build targets in parallel. The **–j** option overrides the environment variable MAKELIM.

–k When a command returns nonzero status, abandon work on the current entry, but continue on branches that do not depend on the current entry.

–S Removes the effect of a previous –k on the command line.

–n Trace and print, but do not execute the commands needed to update the targets.

–q Check if the following target given is up to date, but do not execute commands. Return

exit code 0 if the target is up to date and -1 otherwise.

-r Equivalent to an initial special entry .SUFFIXES: with no list.

-s Equivalent to the special entry .SILENT:.

-t Touch, i.e. update, the modified date of targets, without executing any commands.

-T Like -t, but will not create a file that doesn't already exist.

-v Verbose mode, will print out a message when a target has started building and when it finishes.

-R Equivalent to the special entry .RCSCHECK:.

-W Suppress warnings about *-include* files which cannot be opened.

FILES

makefile
Makefile

SEE ALSO

mkdep(1), sh(1), touch(1), fc(1f)(optional product), cc(1)
"Make – A Program for Maintaining Computer Programs" in the *ConvexOS Tutorial Papers*

DIAGNOSTICS

Exit status of 0 upon successful completion; 1 if no arguments or description file, named rule does not exist, cannot open input or output file, fatal syntax error, cannot get memory; 2 if interrupted during processing and a partial output file is removed.

If the time of modification of a target is either less than or *equal to* a dependent, *make* will build the target whereas previous versions of make would only build the target if its time of modification was less than its dependents.

BUGS

Some commands return nonzero 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*.

Commands that require interactive use will not work if targets are being built in parallel; also to be avoided is building archives in parallel. Specify the **-j** flag with 1 as the argument to have interactive commands work correctly.

make has a limit of 4000 characters for each target name and each dependency line.

The .PREFIXES target has been removed from this version of *make*. The VPATH macro can be used to accomplish the same function.

File targets to be searched on VPATH must be delimited by space, blank, or tab; for instance the shell command:

```
echo foo bar; echo baz
```

in a makefile would find foo and baz on VPATH, but not bar, as it is concatenated to the semicolon with no intervening white space.

The time stamps of files are maintained with one second granularity; this may result in targets being deemed out-of-date with respect to targets when this is not strictly true. For instance, the following makefile fragment:

```
x.s:    x.src
        cpp < x.src > x.s

x.o:    x.s
        as -o x.o x.s
```

may result in the time stamps of x.s and x.o being identical. To cope with this situation during a single invocation of make, the update times of targets are derived from the time at which the rule to produce that target was completed, maintained with microsecond accuracy (see *gettimeofday(2)*). Note that this does not matter when *make* is next invoked, as the two files have the same modification time with one second accuracy.

NAME

man, apropos, whatis – display on-line reference manual information

SYNOPSIS

man [**–ltfgikwuvthdKn**] [**–M** *manpath*] [**-S***sections*] [**-T***troffproc*] [*hwtype*] [*section*] *topic*[*/index*] ...

apropos [**–uvdDK**] [**–M** *manpath*] [*hwtype*] *keyword*

whatis [**–uvdhD**] [**–M** *manpath*] [*hwtype*] *topic* ...

DESCRIPTION

The *man* program (and its links, *apropos* and *whatis*) formats and displays information from the on-line Programmer's Reference Manual. It can display complete entries selected by topic or brief one-line summaries selected by keyword (as *apropos* or with the **–k** flag) or by topic (as *whatis* or with the **–f** flag). When invoked in the simplest way, without any options and with a topic, it displays the corresponding manual page formatted with *nroff*(1). If the standard output is to a terminal, the man page is piped by default through *more*(1), or else through the program specified in the user's PAGER environment variable if set.

Man pages are stored on-line as *nroff* source files and are reformatted whenever the formatted version is missing or has an older modification date than the unformatted version. The formatted version of the man page stored in the corresponding **cat*/** directory if it exists and is writable by the current user. Thus *man1/stty.1* once formatted is stored in *cat1/stty.1*. Otherwise manual pages must be reformatted each time a user wishes to view them. If the root of the man tree from which the man page is taken contains the file *tmac.an*, then this file is used in lieu of the standard *man*(7) macros to reformat the page. Appropriate calls to *tbl*(1) and either *neqn*(1) or *eqn*(1) are automatically inserted as needed.

Section Selection

Some topics occur in more than one section of the manual. To select a man page from a particular section, put the section name in front of the topic, as in `man 4 tty`. For backwards compatibility with other *man* programs, certain non-numeric sections are recognized by longer aliases: **local** means section **l**, **new** means section **n**, **public** means section **p**, and **old** means section **o**. In other words, `man new csh` on the command line will search for a man page on the *csh* in section *n* of the manual.

The applicable topics for each component in the current MANPATH are sorted according to their section. The default ordering is **ln16823457po**. Thus, *stty*(1) would be displayed before *stty*(3). This ordering can be overridden on a site-wide basis by the system manager (see NOTES section below), a per-user basis using the MANSECT environment variable, or a per-command basis using the **-S** *sections* switch. Programmers may prefer a default ordering beginning **231** so that when they type `man wait` they see the man page for the system call rather than the one for the shell command of the same name. If multi-character sections exist (*e.g.* some systems have a **man1m** section) or subsection ordering is desired, then colons must be used to delimit the sections. A FORTRAN programmer might prefer to set his MANSECT to **3f:2:3:1:6:8**, so that when he asks for the manual page for *system*, he gets *system*(3F) rather than the C function of the same name.

Case is significant for the topic itself, but not for the section. This allows you to make a distinction between *cc*(1), the C compiler, and *CC*(1), the C++ translator. However, section **3S** is interpreted as section **3s** to accommodate users who see a reference to the man page for *printf*(3S), and don't realize that it's stored in the file **printf.3s**.

Search Strategy

The *man* program searches by default first in */usr/man* and then in */usr/local/man* for the specified pages. This search order can be overridden on a site-wide basis by the system manager (see NOTES section below) or on a per-user basis by setting the environment variable

MANPATH. This is a colon-delimited list of directory names that contains subdirectories of the form **man*/**. A user keeping his own man pages under his home directory might place a line like the following in his *~/.cshrc* file:

```
setenv MANPATH $HOME/man:/usr/local/man:/usr/man
```

The MANPATH may also be set on a per-command basis by using the **–M** command-line.

It is possible to install several complete sets of man pages on one host. These alternate sets are assumed to be installed in MANALT/*subdir*, where MANALT is an environment variable defaulting to */usr/local/man* and *subdir* is assumed to be a complete man tree. If at least non-switch two arguments are supplied, then the MANALT directory is consulted to see whether it contains a sub-directory whose names is the first argument. This makes it easy to say things like `man sun csh` to read the *csh* man page from the installed set of *sun* man pages. This is equivalent to saying `man -M /usr/local/bin/sun csh` (assuming MANALT is not set).

Indexing

Very long man pages are often broken up into several major subsections, whose tables of contents can be listed with the **–i** switch. To access one of these subsections directly, specify */index*, where */index* is the name of the subsection. For example, to see the section on expressions in the *csh*(1) man page, use `man csh/expr` to skip to that section first. If *index* is not a valid section or subsection header, or if none is given, a list of all valid section headers is displayed and the user is prompted to select one. The special form of a missing subsection after the slash means to go into continuous menu mode, returning to the section menu after the manual page has been displayed.

The match against the index entries is not case-sensitive, nor is it required to be at the beginning of the string. For example, this allows `/expr` to be used to look up the `Regular Expressions` section. However, matches that *do* start at the front of the header are selected before those that do not. A numeric section index as listed by the **–i** flag may also be given, with an index of **0** meaning the whole page.

Man page indices (tables of contents) are generated automatically when needed. To save time, if a writable **idx*/** subdirectory exists in the root of the man tree, the index is left there under the same name as its man page for quicker access in the future. Indices older than their parent man page are rebuilt upon demand.

As a special case of indexing, the form *//string* starts the PAGER at an arbitrary *string* in the text of the man page rather than at a subsection index.

Note that the */index* feature may not be used in conjunction with the **–l** (local file) switch. The **–i** switch may still be used with local files.

OPTIONS

–M*manpath*
: Changes the search path for finding man pages for just this command, overriding the current MANPATH environment variable or system default. See the section on **Search Strategy** for further details.

–S*sections*
: sets the section and subsection ordering for just this command, overriding the current MANSECT environment variables or system default. See the **Section Selection** for further details.

–f directs *man* to print out the applicable one-line descriptions for each corresponding *topic*. These one-line descriptions are stored in separate *whatis* files for each component in the specified MANPATH . If *dbm*(3X) versions of these databases exist, these are used for more rapid location of manual pages. See *makewhatis*(8) for directions on generating the *whatis* databases. This switch is automatically enabled if *man* is invoked as *whatis*.

–g*regexpr*
: direct *man* to search through all man pages for the specified *Perl* regular expression. The

option will work with compressed man pages as well.

–k direct *man* to search the *whatis* databases for any entry matching the given *keyword*. This is useful when the precise topic is unknown; it reports the topics that have to do with the given keywords, according to their *whatis* descriptions. Output will be sent to the user's PAGER. This switch is automatically enabled if *man* is invoked as *apropos*.

–K same as –k except regular expressions as in *egrep*(1) may be used in the *keywords*.

–t is used to typeset the manual pages, usually using *troff*(1L) or a front-end for the same. If the default system typesetter is not desired, it can be overridden either with the TROFF environment variable or the **–T** flag.

–T*troffproc*
selects an alternate typesetting program, overriding the current TROFF environment variable or system default. It is most often used to select a previewer for workstations that support previewing of *troff* output.

–l is used with specific file names rather than with topics. The same processing as *man* would perform on a system manual page will be performed on the given file. This is useful when first writing new manual pages or viewing ones not yet installed. It may be used in conjunction with other switches, such as **–t**. The indexing feature is not available for local man pages except via the –i switch.

–w is used to determine the full pathnames of the files that *man* would display for the given topic. This is useful when the same topic occurs in more than one section of the manual or in more than one of the components in the supplied MANPATH. The files are listed in the order in which they would be presented to the user.

–a displays all man pages for all possible matches for a given topic. Its effect is similar to calling *man* on all the files returned by the **–w** switch.

–i the complete section and subsection index for a given topic.

–v prints out the current revision of the program as stored by *rcs*(1).

–u displays a long usage message, piped through the user's PAGER. It includes the current defaults for PAGER, MANPATH, MANSECT, MANALT, and TROFF.

–h forces *man* to ignore the *whatis* databases and search for its manual pages the slow way. This is the default behavior if no *whatis* database has been created for a particular man tree.

–d triggers debugging output. All externally invoked programs such as *nroff* and *egrep* will have their arguments displayed before they are called. Certain other debugging information is also printed.

–n Also intended for debugging; externally invoked programs such as *nroff* or *eqn* are displayed, but not run.

–D causes embedded backspaces and their preceding characters in cat pages to be stripped upon output. This is useful for producing utterly clean output, such as when being viewed in an *emacs*(1) window.

EXAMPLES

```
man stty                    # show me the first stty man page
man -w stty                 # which stty man pages are there?
man 3 stty                  # show me the one from section 3
man -a tty                  # show all man pages on tty

man old makewhatis          # show me the makewhatis man page from mano
man new csh                 # show me the csh man page from mann
```

```
man sun csh                          # show me the csh man page for suns
                                     (assumes $MANALT/sun contains sun man pages)

man -i uucp                          # get list of sections in uucp man page
man -ai tty                          # give list of sections on all tty pages
man adb/bugs                         # check for BUGS section of adb man page
man man/exam                         # start at this example section
man ksh/                             # select from section menu for ksh man page
man sun cron/files                   # go to the FILES section of the sun cron man page

man -f rcs                           # what is rcs?
whatis rcs                           # same as previous
whatis vax rcs                       # same as previous, but only vax man pages

man -k rcs                           # what knows about rcs?
apropos rcs                          # same as previous
apropos sun rcs                      # same as previous, but only sun man pages

man -S231 wait                       # override system section ordering
man -S3f:3s:3:2:1 system             # subsection ordering

man -M ~/man hack                    # use ~/man as only man tree

man -t 4 tty                         # troff tty(4) man page
man -at tty                          # troff all tty pages
man -Tpnitroff perl                  # preview perl man page
man -lTpnitroff ../foo.1             # preview local file as man page

man -l file.x                        # run man on local file
man -tl file.x                       # print local file as man would
man -il file.x                       # get section index on local page
```

ENVIRONMENT

Man explicitly checks for the following environment variables; if they are present, they will override its internal defaults:

MANPATH The colon-delimited list of man trees for locating man pages. This may itself be overridden by the command line flags **–M** or by specifying an alternate hardware type.

MANSECT The default ordering for section and subsection sorting. It need only be delimited by colons if multi-character sections such as **man1m** exist or if subsection ordering is desired, such as to place subsection **3f** in front of section **3 .**

MANALT The directory to check for alternate sets of man trees. This is useful for storing the man pages for several different machine architectures or versions of the operating system. See the **Search Strategy** section for details.

PAGER The user's preferred viewer of paged output. Note that often the pager itself will consult the environment; for example, *more* looks for a MORE variable and *less* looks for a LESS variable in the environment.

TROFF The preferred typesetter program. It should recognize *troff* input and switches. This may be overridden using the **–T** command line flag.

FILES

/usr/man	default man tree
/usr/man/man/*.**	unformatted (nroff source) man pages

/usr/man/cat/*.**	formatted man pages
/usr/man/idx/*.**	indices for section headers
/usr/man/whatis	default whatis database, text version
/usr/man/whatis.dir	*dbm* index file for default *whatis* database
/usr/man/whatis.pag	*dbm* data file for default *whatis* database
/usr/lib/perl/getopts.pl	required *perl* library file
/usr/lib/perl/stat.pl	required *perl* library file

SEE ALSO

egrep(1), *perl*(1), *more*(1), *eqn*(1), *tbl*(1), *nroff*(1), *troff*(1L), *compress*(1L), *dbm*(3X), *man*(7), *catman*(8), *makewhatis*(8)

NOTES

This version of *man* is written entirely in the *perl*(1) programming language and thus requires that *perl* be properly installed on your system to run. Because *man* is distributed in a script, it can be easily reconfigured by the system managers to suit their site's particular style. This is good, as some of the default settings are somewhat idiosyncratic to the set-up at the CONVEX home office.

Be sure to save a copy of the original *man* program before any modification.

Things that can be configured include:

- the default PAGER and its flags (alternate flags can be provided for *less*)
- system defaults for MANPATH, MANSECT, MANALT, and TROFF.
- paths for *troff, nroff, tbl, eqn, neqn, ul, col, egrep,* and *compress*
- whether you have compressed man pages and how they are stored
- which section aliases you want (as in **public** being recognized as section **p**)
- whether to recognize man pages whose NAME sections don't mention their own names

To save disk space at the expense of execution time, a site may wish to run *compress*(1L) on the manual entries where available. The *man* program understands how to read compressed man pages, and knows to create a compressed cat page if the source man page was compressed to start with.

When running on slow CPUs, the start-up time for parsing the script may be annoying. These sites can skip this parsing phase at each invocation of *man*(1) by using *perl*'s –u flag to dump a binary image of the interpreter.

DIAGNOSTICS

Several self-explanatory diagnostics are possible, such as `No manual entry for xyzzy`, but the following warnings may not be intuitive:

`But what do you want from section %s?`
A section was specified but no topic.

`No dbm file for %s: %m`
This means that the named man tree does not have an accessible *dbm* version of its *whatis* database and that *makewhatis*(8) should be run on it. This only shows up when –d option is used. %m is the appropriate *perror*(3) message.

`%s has disappeared -- rerun makewhatis`
A man page has been removed since *makewhatis* was last run. Note that new man pages being added will NOT be detected.

nroff of %s failed
For some reason *nroff* did not exit correctly. The disk may be mounted read-only, it might be full, or *nroff* may not be installed on your system. Contact your system manager.

%s was length 0; disk full?
A cat page was empty. The *man* program unlinks the useless cat page, issues this warning, and exits non-zero. Rerun the command and if the problem persists, contact your system manager.

/tmp not writable
The */tmp* and */usr/man/cat** directories are not writable by you. Contact your system manager.

No whatis databases found, please run makewhatis
The *man* program was unable to find a *whatis* database for use with *apropos* or *man -k* in any of the directories listed in the **MANPATH**. Have your system manager run *makewhatis* on the system manual page directories, and run *makewhatis* on any personal manual page directories.

bad eval: %s
can't eval %s: %s
These are internal errors that should never occur. Contact your system manager, who should submit a problem report.

RESTRICTIONS

Once a *dbm*(3X) *whatis* database has been created for a particular man root, new man pages will not be found unless the **-h** flag is used or until *makewhatis* is rerun.

If your **PAGER** is *more*(1), bold text shows up in the normal font; however, *less*(1) does display bold text in your terminal's bold font.

BUGS

The manual is supposed to be reproducible either on the phototypesetter or on an ASCII terminal. However, on a terminal, some information is necessarily lost.

The **-t** flag for *man* only works if the host system supports *troff*.

Not all systems have *compress* installed on them.

AUTHOR

Tom Christiansen <*tchrist@ convex.com*>

COPYRIGHT

NAME

merge – three-way file merge

SYNOPSIS

merge [**-p**] file1 file2 file3

DESCRIPTION

Merge incorporates all changes that lead from *file2* to *file3* into *file1*. The result goes to std. output if **-p** is present, into *file1* otherwise. *Merge* is useful for combining separate changes to an original. Suppose *file2* is the original, and both *file1* and *file3* are modifications of *file2*. Then *merge* combines both changes.

An overlap occurs if both *file1* and *file3* have changes in a common segment of lines. *Merge* prints how many overlaps occurred, and includes both alternatives in the result. The alternatives are delimited as follows:

```
<<<<<<< file1
lines in file1
=======
lines in file3
>>>>>>> file3
```

If there are overlaps, the user should edit the result and delete one of the alternatives.

IDENTIFICATION

Author: Walter F. Tichy,
Revision Number: 0.7 ; Release Date: 91/10/08 .

SEE ALSO

diff3 (1), diff (1), rcsmerge (1), co (1).

NAME

mesg – permit or deny messages

SYNOPSIS

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

DESCRIPTION

Mesg with argument **n** forbids messages via *write* and *talk*(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), talk(1)

DIAGNOSTICS

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

NAME

mkdep – construct Makefile dependency list

SYNOPSIS

mkdep [**–a**] [**–c**] [**–f makefile**] [**–l library**] [**–p**] [**–t**] [**-C compiler**] [flags] file ...

DESCRIPTION

Mkdep takes a set of flags for the C compiler and a list of C source files as arguments and constructs a set of include file dependencies. It attaches this dependency list to the end of the file "Makefile". An example of its use in a makefile might be:

```
CFLAGS= -O -I../include -I.
SRCS= file1.c file2.c

depend:
        mkdep $(CFLAGS) $(SRCS)
```

where the macro SRCS is the list of C source files and the macro CFLAGS is the list of flags for the C compiler. If the **-c** option is specified, *mkdep* attaches a ".c" suffix to each source file name before constructing the dependencies. The **-f** option provides *mkdep* with a name other than "Makefile" to be edited. If the **-p** option is provided, *mkdep* produces dependencies of the form

```
program: program.c
```

so that subsequent makes will produce *program* directly from its C module rather than using an intermediate *.o* module. This is useful in directories that contain many programs, each of whose source is contained in a single C module. If the **-C** option is specified, *mkdep* expects **compiler** to contain the pathname of a compiler that accepts the **-k** argument. (See cc(1) for more information.) The default compiler is */bin/cc*.

The **-l library** flag specifies that dependency lines are to be written using the object library string specified in the following argument; the argument string is prepended to the object name and the object name is parenthisized. For instance,

```
mkdep -l '$(LIB)'
```

might produce lines which look like

```
$(LIB)(file.o): file.c /usr/include/stdio.h
```

The **-t** flag is used to preserve the modification times of the Makefile even though it is actually updated. This is designed for those rare circumstances when objects have explicit dependencies on the Makefile, yet no significant change has been made to the Makefile.

The **-a** flag causes mkdep to append new dependencies without first removing old dependencies; this can be useful when different files processed by the same Makefile require distinct compiler flags. In this case, the "depend" rule in the Makefile might appear:

```
depend:
        mkdep -c -p $(PROG1FLAGS) prog1
        mkdep -a -c -p $(PROG2FLAGS) prog2
```

SEE ALSO

cc(1), make(1), touch(1)

NAME

mkdir – make a directory

SYNOPSIS

mkdir [**–fp**] dirname ...

DESCRIPTION

mkdir creates specified directories in mode 777. Standard entries, '.', for the directory itself, and '..' for its parent, are made automatically.

mkdir requires write permission in the parent directory.

The maximum file or directory name length is defined in *<sys/dir.h>* by the constant MAXNAMLEN. The maximum pathname length (after expanding symbolic lengths) is defined in *<sys/param.h>* by the constant MAXPATHLEN (minus one for the null termination character).

There are two options:

–p If *dirname* is a pathname and this option is specified, all directories in the pathname are created if they do not already exist.

–f Suppresses error message if *dirname* already exists.

SEE ALSO

rm(1)

NAME

mkstr – create an error message file by massaging C source

SYNOPSIS

mkstr [–] messagefile prefix file ...

DESCRIPTION

Mkstr is used to create files of error messages. Its use can make programs with large numbers of error diagnostics much smaller, and reduce system overhead in running the program as the error messages do not have to be constantly swapped in and out.

Mkstr will process each of the specified *files*, placing a massaged version of the input file in a file whose name consists of the specified *prefix* and the original name. A typical usage of *mkstr* would be

```
mkstr pistrings xx *.c
```

This command would cause all the error messages from the C source files in the current directory to be placed in the file *pistrings* and processed copies of the source for these files to be placed in files whose names are prefixed with *xx*.

To process the error messages in the source to the message file *mkstr* keys on the string 'error("' in the input stream. Each time it occurs, the C string starting at the '"' is placed in the message file followed by a null character and a new-line character; the null character terminates the message so it can be easily used when retrieved, the new-line character makes it possible to sensibly *cat* the error message file to see its contents. The massaged copy of the input file then contains a *lseek* pointer into the file which can be used to retrieve the message, i.e.:

```
char    efilname[] = "/usr/lib/pi_strings";
int     efil = -1;

error(a1, a2, a3, a4)
{
        char buf[256];

        if (efil < 0) {
                efil = open(efilname, 0);
                if (efil < 0) {
oops:
                        perror(efilname);
                        exit(1);
                }
        }
        if (lseek(efil, (long) a1, 0) < 0 || read(efil, buf, 256) <= 0)
                goto oops;
        printf(buf, a2, a3, a4);
}
```

The optional – causes the error messages to be placed at the end of the specified message file for recompiling part of a large *mkstr* ed program.

SEE ALSO

lseek(2), xstr(1)

AUTHORS

William Joy and Charles Haley

NAME

more, page – file perusal filter for CRT viewing

SYNOPSIS

more [**–cdflsu**] [*–n*] [*+linenumber*] [*+/pattern*] [*filename ...*]

page *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 hits a carriage return, one more line is displayed. If the user hits a space, 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 an editor at the current line. The default editor is *vi*(1). This can be changed by setting one of the environment variables *MOREED* or *EDITOR*. The way that *more* instructs the editor to begin editing at the current line is to invoke the editor by "editor +n filename", where n is the current line number. If *more* finds an editor in *MOREED*, *more* will instead invoke the editor by "editor filename". Thus, if your editor cannot be invoked by "editor +n filename" you should use *MOREED*. Otherwise, to specify an editor which is not the default editor *vi*(1), you should use *EDITOR*.

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.

t
: Go to the top of the current file.

!command
: Invoke a shell with *command*. The characters '%' and '!' in "command" are replaced with the current filename and the previous shell command respectively. If there is no current filename, '%' 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 *i* doesn't make sense).

i:p Skip to the *i*-th previous file given in the command line. If this command is given in the middle of printing out a file, then *more* goes back to the beginning of the file. If *i* doesn't make sense, *more* skips back to the first file. If *more* is not reading from a file, the bell is rung and nothing else happens.

:f Display the current file name and line number.

:q or :Q
Exit from *more* (same as q or Q).

. (dot) Repeat the previous command.

The commands take effect immediately, i.e., it is not necessary to type a carriage return. Up to the time when the command character itself is given, the user may hit the line kill character to cancel the numerical argument being formed. In addition, the user may hit the erase character to redisplay the "--More--(xx%)" message.

At any time when output is being sent to the terminal, the user can hit the quit key (normally control-\). *More* will stop sending output, and will display the usual "--More--" prompt. The user may then enter one of the above commands in the normal manner. Unfortunately, some output is lost when this is done, due to the fact that any characters waiting in the terminal's output queue are flushed when the quit signal occurs.

The terminal is set to *noecho* mode by this program so that the output can be continuous. What you type will thus not show on your terminal, except for the / and ! commands.

If the standard output is not a teletype, then *more* acts just like *cat*, except that a header is printed before each file (if there is more than one).

A sample usage of *more* in previewing *nroff* output would be:

```
nroff -ms +2 doc.n | more -s
```

FILES

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

SEE ALSO

csh(1), man(1), msgs(1), script(1), sh(1), environ(7)

NAME

mpa – modify process attributes

SYNOPSIS

mpa [**–f** | **–d** | **–v**] [**–t***nthreads*] [**–c***cpunumber*] command [arguments]

mpa [**–f** | **–d** | **–v**] [**–c***cpunumber*] **–p***pid* ...

DESCRIPTION

mpa is used to modify a specified *command* or pids attributes. The currently supported process attributes are associated with type of process scheduling. The following scheduling options are;

–f Changes process or pid to run as a fixed schedule job. A fixed schedule job is always scheduled with all CPUs available for it's use.

–d Changes process or pid to run as a dynamic schedule job. A dynamic schedule job is scheduled on one or more CPUs based on availability any time the process is reschedule.

–v Changes process or pid to run with the process virtual timer enabled. The process virtual timer cannot be enabled while the process is multithreaded (multiple active threads).

–t*nthreads*
Specifies the maximum number of threads the process can use during it's execution. This switch is only meaningful if a *command* argument is specified.

–c*cpunumber*
Specifies which cpu a process will run on during execution. This switch only works on single-threaded processes. *mpa* sets the maximum number of threads to one and sets the scheduling attribute to be dynamic when this option is selected. Only root can set cpu affinity.

The default schedule for all processes is dynamic unless changed with *mpa*. An already executing process's attributes may be changed by using the **–p** switch to specify that arguments to *mpa* are process ids instead of a *command*. If **–p** is used without specifying a change in the process attributes, the processes current attributes are displayed.

A user may only change the process's attributes if the caller has the same uid as the process to be changed. User id 0, root, may change any other process's attributes.

SEE ALSO

getrlimit(2), setrlimit(2), setpattr(2), getpattr(2), acct(8)

DIAGNOSTICS

mpa returns the exit status of the subject command.

BUGS

NAME

msgs – system messages and junk mail program

SYNOPSIS

msgs [**fhlopqs**] [*number*] [*-number*]

msgs -c [*-days*]

DESCRIPTION

Msgs is used to read system messages. These messages are sent by mailing to the login 'msgs' and should be short pieces of information which are suitable to be read once by most users of the system.

Msgs is normally invoked each time you login, by placing it in the file *.login (.profile* if you use */bin/sh*). It will then prompt you with the source and subject of each new message. If there is no subject line, the first few non-blank lines of the message will be displayed. If there is more to the message, you will be told how long it is and asked whether you wish to see the rest of the message. The possible responses are:

y Type the rest of the message.

RETURN Synonym for y.

n Skip this message and go on to the next message.

p (P) Type the rest of the message and turn on (off) piping through *more*(1) for this and subsequent messages. Overrides the **-p** command line option.

– Redisplay the last message.

q Drops you out of *msgs;* the next time you run the program it will pick up where you left off.

q Immediate exit from *msgs.*

s Append the current message to the file "Messages" in the current directory; 's–' will save the previously displayed message. A 's' or 's–' may be followed by a space and a filename to receive the message replacing the default "Messages".

m Or 'm–' causes a copy of the specified message to be placed in a temporary mailbox and *mail*(1) to be invoked on that mailbox. Both 'm' and 's' accept a numeric argument in place of the '–'.

Msgs keeps track of the next message you will see by a number in the file *.msgsrc* in your home directory. In the directory */usr/msgs* it keeps a set of files whose names are the (sequential) numbers of the messages they represent. The file */usr/msgs/bounds* shows the low and high number of the messages in the directory so that *msgs* can quickly determine if there are no messages for you. If the contents of *bounds* is incorrect it can be fixed by removing it; *msgs* will make a new *bounds* file the next time it is run.

Options to msgs include:

-c [*-days*] Remove messages that were written more than a given number of days ago. The default is 21 days. Only the superuser or daemon can use this option.

–f Causes *msgs* not to say "No new messages.". This is useful in your *.login* file since often there are no new messages when logging in.

–h Causes *msgs* to print the first part of messages only.

–l 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.

–o Causes *msgs* to give a prompt after the last message is typed, instead of exiting automatically.

-p Causes long messages to be piped through *more*(1).

-q Queries whether there are messages, printing "There are new messages." if there are. The command "msgs -q" is often used in login scripts.

-s Write a new message.

-*number* Causes *msgs* to start *number* messages back from the one indicated by your *.msgsrc* file, useful for reviews of recent messages.

Within *msgs* you can also go to any specific message by typing its number when *msgs* requests input as to what to do.

FILES

/usr/msgs/*	database
~/.msgsrc	number of next message to be presented

AUTHORS

William Joy
David Wasley

SEE ALSO

mail(1), more(1)

BUGS

NAME

mt – magnetic tape manipulating program

SYNOPSIS

mt [**–f** *tapename*] *command* [*count*]

DESCRIPTION

mt is used to give commands to a magnetic tape drive. If *tapename* is not specified, the environment variable TAPE is used; if TAPE does not exist, *mt* queries *tpdaemon* for the tape device; if that fails, *mt* uses the device */dev/rmt12*. Note that *tapename* must reference a raw (not block) tape device. By default *mt* performs the requested operation once. Operations may be performed multiple times by specifying *count*.

The available *command*(s) are listed below. Only as many characters as are required to uniquely identify a *command* need be specified.

eof, weof
Write *count* end-of-file marks at the current position on the tape.

fsf Forward space *count* files. For labeled tapes, this command (and the analogous *bsf* command) treats all the header and trailer labels and file data associated with a tape file as a single unit of movement.

fsr Forward space *count* records. For labeled tapes, the record of movement used for this command is a logical record (e.g., a line of data), which is not necessarily a physical tape record.

bsf Back space *count* files. This command performs the same function as the MTBSF operation described in *mtio*(4); it moves the tape backwards until *count* tape marks have been crossed. This operation always leaves the tape positioned immediately *before* an EOF, so the next file read after this command will appear to be empty. To back up to the beginning of the previous file on the tape, issue **mt bsf 2** followed by **mt fsf**.

bsr Back space *count* records. This command is not supported for labeled tapes.

rewind
Rewind the tape (*Count* is ignored.) For multi-volume labeled tapesets, this command has the effect of rewinding the entire tapeset (i.e., all physical tapes are treated as a single long logical tape) back to the beginning of the first volume in the set.

offline, rewoffl
Rewind the tape and place the tape unit offline (*Count* is ignored.) For labeled tapes, this command is invalid and the user is requested to use *tpunmount*(1) instead.

gap Write *count* extended interrecord gaps at the current position on the tape.

clear, clr
Clear the drive slave status register.

status Print status information about the tape unit.

mt returns a 0 exit status if the operation(s) are successful, 1 if the command was unrecognized, and 2 if an operation failed.

RESTRICTIONS

mt fsr 1 cannot be used to skip over the tape mark between files.

Tape operations must be performed on *raw* tape devices not *block* tape devices.

Regardless of the tape operation specified, tape operations performed on *rewind-on-close* tape devices always result in the tape being rewound and positioned at the beginning of tape. This occurs because after the tape operation, the tape device is closed.

The *bsr* (back space record) command is not supported for labeled tapes (and therefore has no effect on the tape position).

FILES

*/dev/rmt** Raw magnetic tape interface

SEE ALSO

mtio(4), dd(1), ioctl(2), environ(7), tpmount(1)

NAME

mv – move or rename files

SYNOPSIS

mv [**–i**] [**–f**] [–] file1 file2

mv [**–i**] [**–f**] [–] file ... directory

DESCRIPTION

Mv moves (changes the name of) *file1* to *file2*.

If *file2* already exists, it is removed before *file1* is moved. If *file2* has a mode which forbids writing, *mv* prints the mode (see *chmod*(2)) and reads the standard input to obtain a line; if the line begins with **y**, the move takes place; if not, *mv* exits.

In the second form, one or more *files* (plain files or directories) are moved to the *directory* with their original file-names.

Mv refuses to move a file onto itself.

Options:

–i stands for interactive mode. Whenever a move is to supercede an existing file, the user is prompted by the name of the file followed by a question mark. If he answers with a line starting with 'y', the move continues. Any other reply prevents the move from occurring.

–f stands for force. This option overrides any mode restrictions or the –i switch.

– means interpret all the following arguments to *mv* as file names. This allows file names starting with minus.

SEE ALSO

cp(1), ln(1)

BUGS

If *file1* and *file2* 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.

NAME

neqn, checkeq – typeset mathematics

SYNOPSIS

neqn [**–dxy**] [**–pn**] [**–sn**] [**–fn**] [file] ...
checkeq [file] ...

DESCRIPTION

Neqn is an nroff(**1**) preprocessor for typesetting mathematics on terminals. Usage is almost always

neqn file ... | nroff

If no files are specified, these programs reads from the standard input. A line beginning with '.EQ' marks the start of an equation; the end of an equation is marked by a line beginning with '.EN'. Neither of these lines is altered, so they may be defined in macro packages to get centering, numbering, etc. It is also possible to set two characters as 'delimiters'; subsequent text between delimiters is also treated as *neqn* input. Delimiters may be set to characters *x* and *y* with the command-line argument **–d***xy* or (more commonly) with 'delim *xy*' between .EQ and .EN. The left and right delimiters may be identical. Delimiters are turned off by 'delim off'. All text that is neither between delimiters nor between .EQ and .EN is passed through untouched.

The program *checkeq* reports missing or unbalanced delimiters and .EQ/.EN pairs.

Tokens within *neqn* are separated by spaces, tabs, newlines, braces, double quotes, tildes or circumflexes. Braces {} are used for grouping; generally speaking, anywhere a single character like *x* could appear, a complicated construction enclosed in braces may be used instead. Tilde ~ represents a full space in the output, circumflex ^ half as much.

Subscripts and superscripts are produced with the keywords **sub** and **sup.** Thus *x sub i* makes $x sub i$, *a sub i sup 2* produces $a sub i sup 2$, and *e sup {x sup 2 + y sup 2}* gives $e sup {x sup 2 + y sup 2}$.

Fractions are made with **over**: *a over b* yields $a over b$.

sqrt makes square roots: *1 over sqrt {ax sup 2 +bx+c}* results in $1 over sqrt {ax sup 2 +bx+c}$.

The keywords **from** and **to** introduce lower and upper limits on arbitrary things: $lim from {n-> inf} sum from 0 to n x sub i$ is made with *lim from {n-> inf} sum from 0 to n x sub i.*

Left and right brackets, braces, etc., of the right height are made with **left** and **right:** *left [x sup 2 + y sup 2 over alpha right] ~=~1* produces $left [x sup 2 + y sup 2 over alpha right] ~=~1$. The **right** clause is optional. Legal characters after **left** and **right** are braces, brackets, bars, **c** and **f** for ceiling and floor, and "" for nothing at all (useful for a right-side-only bracket).

Vertical piles of things are made with **pile**, **lpile**, **cpile**, and **rpile**: *pile {a above b above c}* produces $pile {a above b above c}$. There can be an arbitrary number of elements in a pile. **lpile** left-justifies, **pile** and **cpile** center, with different vertical spacing, and **rpile** right justifies.

Matrices are made with **matrix**: *matrix { lcol { x sub i above y sub 2 } ccol { 1 above 2 } }* produces $matrix { lcol { x sub i above y sub 2 } ccol { 1 above 2 } }$. In addition, there is **rcol** for a right-justified column.

Diacritical marks are made with **dot**, **dotdot**, **hat**, **tilde**, **bar**, **vec**, **dyad**, and **under**: *x dot = f(t) bar* is $x dot = f(t) bar$, *y dotdot bar ~=~ n under* is $y dotdot bar ~=~ n under$, and *x vec ~=~ y dyad* is $x vec ~=~ y dyad$.

Sizes and font can be changed with **size** *n* or **size** ± *n*, **roman**, **italic**, **bold**, and **font** *n*. Size and fonts can be changed globally in a document by **gsize** *n* and **gfont** *n*, or by the command-line arguments **–s***n* and **–f***n*.

Normally subscripts and superscripts are reduced by 3 point sizes from the previous size; this may be changed by the command-line argument **-p***n*.

Successive display arguments can be lined up. Place **mark** before the desired lineup point in the first equation; place **lineup** at the place that is to line up vertically in subsequent equations.

Shorthands may be defined or existing keywords redefined with **define**: *define thing % replacement %* defines a new token called *thing* which will be replaced by *replacement* whenever it appears thereafter. The *%* may be any character that does not occur in *replacement*.

Keywords like *sum* (sum) *int* (int) *inf* (inf) and shorthands like >= (>=) -> (->), and != (!=) are recognized. Greek letters are spelled out in the desired case, as in *alpha* or *GAMMA*. Mathematical words like sin, cos, log are made Roman automatically. *Troff*(1) four-character escapes like \(bs (\) can be used anywhere. Strings enclosed in double quotes "..." are passed through untouched; this permits keywords to be entered as text, and can be used to communicate with *troff* when all else fails.

SEE ALSO

troff(1)(optional product), tbl(1), ms(7), eqnchar(7)
B. W. Kernighan and L. L. Cherry, *A System for Typesetting Mathematics*
J. F. Ossanna, *NROFF/TROFF User's Manual*
(Both of these documents are included in the *ConvexOS Tutorial Papers*.)

BUGS

To embolden digits, parens, etc., it is necessary to quote them, as in 'bold "12.3"'.

Output looks best when routed to a letter-quality printer.

NAME

netstat – show network status

SYNOPSIS

netstat [**–Aan**] [**–f** *address_family*] [*system*] [*core*]
netstat [**–imnrstu**] [**–f** *address_family*] [*system*] [*core*]
netstat [**–n**] [**–I** *interface*] *interval* [*system*] [*core*]
netstat [**–p** *protocol*] [*system*] [*core*]

DESCRIPTION

The *netstat* command symbolically displays the contents of various network-related data structures. There are a number of output formats, depending on the options for the information presented. The first form of the command displays a list of active sockets for each protocol. The second form presents the contents of one of the other network data structures according to the option selected. Using the third form, with an *interval* specified, *netstat* continuously displays the information regarding packet traffic on the configured network interfaces. The fourth form displays statistics about the named protocol.

The options have the following meanings:

–A With the default display, show the address of any protocol control blocks associated with sockets; used for debugging.

–a With the default display, show the state of all sockets; normally sockets used by server processes are not shown.

–d With either interface display (option **–i** or an interval, as described below), show the number of dropped packets.

–f *address_family*
Limit statistics or address control block reports to those of the specified *address family*. The following address families are recognized: *inet*, for **AF_INET**, and *unix*, for **AF_UNIX**.

–i Show the state of interfaces which have been auto-configured (interfaces statically configured into a system, but not located at boot time are not shown).

–I *interface*
Show information only about this interface; used with an *interval* as described below.

–m Show statistics recorded by the memory-management routines (the network manages a private pool of memory buffers).

–n Show network addresses as numbers (normally *netstat* interprets addresses and attempts to display them symbolically). This option may be used with any of the display formats.

–p *protocol*
Show statistics about *protocol*, which is either a well-known name for a protocol or an alias for it. Some protocol names and aliases are listed in the file */etc/protocols*. A null response typically means that there are no interesting numbers to report. The program will complain if *protocol* is unknown or if there is no statistics routine for it.

–r Show the routing tables. When **–s** is also present, show routing statistics instead.

–s Show per-protocol statistics.

–t When the **-i** flag is set, the **-t** flag causes the network interval timer to be displayed instead of the number of collisions.

–u Display UNIX domain information. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

The arguments *system* and *core* allow substitutes for the defaults "/vmunix" and "/dev/kmem".

The default display, for active sockets, shows the local and remote addresses, send and receive queue sizes (in bytes), protocol, and the internal state of the protocol. Address formats are of the form "host.port" or "network.port" if a socket's address specifies a network but no specific host address. When known, the host and network addresses are displayed symbolically according to the data bases */etc/hosts* and */etc/networks*, respectively. If a symbolic name for an address is unknown, or if the **–n** option is specified, the address is printed numerically, according to the address family. For more information regarding the Internet "dot format," refer to *inet*(3N). Unspecified, or "wildcard," addresses and ports appear as "*".

The interface display provides a table of cumulative statistics regarding packets transferred, errors, and collisions. The network addresses of the interface and the maximum transmission unit ("mtu") are also displayed. An interface name with an "*" next to it is down.

The routing table display indicates the available routes and their status. Each route consists of a destination host or network and a gateway to use in forwarding packets. The flags field shows the state of the route ("U" if "up"), whether the route is to a gateway ("G"), whether the route is for a particular host ("H"), whether the route was created dynamically by a redirect ("D"), and whether the route has been modified by a redirect ("M"). Direct routes are created for each interface attached to the local host; the gateway field for such entries shows the address of the outgoing interface. The refcnt field gives the current number of active uses of the route. Connection oriented protocols normally hold on to a single route for the duration of a connection while connectionless protocols obtain a route while sending to the same destination. The use field provides a count of the number of packets sent using that route. The interface entry indicates the network interface utilized for the route.

When *netstat* is invoked with an *interval* argument, it displays a running count of statistics related to network interfaces. This display consists of a column for the primary interface (the first interface found during autoconfiguration) and a column summarizing information for all interfaces. The primary interface may be replaced with another interface with the *–I* option. The first line of each screen of information contains a summary since the system was last rebooted. Subsequent lines of output show values accumulated over the preceding interval.

FILES

/vmunix	system namelist
/dev/mem	kernel memory
*/lib/kernsyms/symdata_**	kernel symbol addresses

SEE ALSO

iostat(1), *vmstat*(1), *hosts*(5), *networks*(5), *protocols*(5), *services*(5), *trpt*(8C)

BUGS

The notion of errors is ill-defined. Collisions mean something else for the IMP.

NOTES

netstat is an optional product; for more information, contact your CONVEX sales representative.

NAME

newaliases – rebuild the data base for the mail aliases file

SYNOPSIS

newaliases

DESCRIPTION

Newaliases rebuilds the random access data base for the mail aliases file */usr/lib/aliases*.

NOTE: It no longer needs be run each time */usr/lib/aliases* is changed in order for the change to take effect. This is now handled automatically by *sendmail*.

SEE ALSO

aliases(5), sendmail(8)

BUGS

NAME

nfcount – print number of notes with/without director's messages in a notesfile

SYNOPSIS

nfcount [**-d**] *topic* [...]

DESCRIPTION

nfcount gives the user a count of the number of notes without director's messages in the named notesfiles. *nfcount* also prints the total number of notes in the notesfile. *nfcount* writes to standard output.

The **-d** parameter changes the output to print the number of notes with director's messages in the named notesfiles.

The *topic* is the set of notesfiles which are to be examined. An example topic is "net.bugs*".

FILES

/usr/spool/notes the default notesfile data base

SEE ALSO

notes(1),
"Notesfile Reference Manual" in the *ConvexOS Tutorial Papers*

NAME

nfpipe – insert articles into a notesfile

SYNOPSIS

nfpipe topic [**-t** title] [**-d**] [**-a**]

DESCRIPTION

nfpipe reads standard input to create a note in the specified notesfile. The title of the note is specified using the **-t** parameter. The **-d** flag specifies that this note is to have the director message flag set; if the author has no director privileges in the notesfile, this flag has no effect. Specifying **-a** makes the note anonymous; if anonymous notes are not permitted this flag has no effect.

FILES

/etc/passwd	for the users name
/etc/group	for the users group
/usr/spool/notes	the default notesfile data base
/usr/spool/notes/.utilities	utility programs and online help the permission list for the notesfile "topic"

SEE ALSO

notes(1), nfcomment(3), nfmail(8),
"The Notesfile Reference Manual" in the *ConvexOS Tutorial Papers*

NAME

nfprint – print the contents of a notesfile

SYNOPSIS

nfprint [**-p**] [**-l#**] [**-d** or **-nd**] [**-c**] [**-t**] topic [*note-list*]

DESCRIPTION

nfprint gives the user the ability to print the contents of notesfiles. **nfprint** writes to standard output. The text is formatted with *pr*(1). When the **-c** option is used, the output is filtered through *cat*(1) instead.

The **-l#** parameter specifies the page length to use (66 lines per page is the default). By specifying **-p**, the printout is arranged so each base note starts on a new page. The **-d** and **-nd** options specify only notes with the *director flag* on or off respectively are to be printed. Use **-t** to generate a list of titles only; the text of notes and responses is suppressed.

The **note list** is the set of notes which are to be printed. An example note list is: 1,30-36,13,10,42-50.

FILES

/bin/pr	output filter
/etc/passwd	for the users name
/etc/group	for the users group(s)
/usr/spool/notes	default notesfile data base

SEE ALSO

notes(1), pr(1),
"Notesfile Reference Manual" in the *ConvexOS Tutorial Papers*

NAME

nfstats – print statistics about Notesfiles

SYNOPSIS

nfstats [**-s**] notefile [...]

DESCRIPTION

nfstats prints usage statistics for notesfiles. This program takes a list of notesfiles and lists their statistics on standard output. If more than one notesfile is specified, an aggregate set of statistics is also produced.

Specify **-s** to have only this aggregate report printed and the individual reports suppressed.

BUGS

Notes sent to two sites are counted as "networked out" twice.

Sequencing past an idle notesfile isn't counted as an entry. This is a mixed blessing. Not keeping this statistic greatly improves performance.

FILES

/etc/passwd	for the user's name
/etc/group	for the user's group
/usr/spool/notes	the default notesfile data base

SEE ALSO

notes(1), nfcomment(3),
"The Notesfile Reference Manual" in the *ConvexOS Tutorial Papers*

NAME

nice, nohup – run a command at low priority (sh only)

SYNOPSIS

nice [*–number*] command [arguments]

nohup command [arguments]

DESCRIPTION

nice executes *command* with low scheduling priority. If the *number* argument is present, the priority is incremented (higher numbers mean lower priorities) by that amount up to a limit of 64. The default *number* is 10.

The superuser may run commands with priority higher than normal by using a negative priority (e.g. '–-10'). However, when running commands with a very negative nice value, other tasks will receive a smaller portion of the cpu. This may result in degradation of overall system performance.

nohup executes *command* immune to hangup and terminate signals from the controlling terminal. The priority is incremented by 5. *nohup* should be invoked from the shell with *&* in order to prevent it from responding to interrupts by or stealing the input from the next person who logs in on the same terminal. The syntax of *nice* is also different.

FILES

nohup.out standard output and standard error file under *nohup*

SEE ALSO

csh(1), getpriority(2), renice(8)

DIAGNOSTICS

nice returns the exit status of the subject command.

BUGS

nice and *nohup* are particular to *sh*(1). If you use *csh*(1), commands executed with *&* are automatically immune to hangup signals while in the background. There is a builtin command (*nohup*) which provides immunity from terminate, but it does not redirect output to *nohup.out*.

nice is built into *csh*(1) with a slightly different syntax than described here. The form "nice +10" nices to positive nice, and "nice –10" can be used by the superuser to give a process more of the processor.

NAME

nm – print name list

SYNOPSIS

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 is 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), **S** (initialized common block), **f** (filename), or **–** for *csd* symbol table entries (see **–a** below). If the symbol is local (nonexternal), the type letter is in lowercase. If the symbol is a unique thread symbol, the type letter will be preceded by the letter **t**. The output is sorted alphabetically.

Options are:

–a Print all symbols; even *csd* 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(5), a.out(5), stab(5)

NAME

notes, autoseq, readnotes – a news system

SYNOPSIS

notes [**–sxin**] [**-o** *date-spec*] [**-a** subsequencer] [**–t** termtype] [**-f** file] topic1 [...]

autoseq [**-a** subsequencer]

readnotes [**-a** subsequencer]

DESCRIPTION

Notes supports computer managed discussion forums. It coordinates access to and updates of data bases of notes and their responses. A single *notesfile* contains an ordered list of *base notes*, each of which may have an ordered list of responses associated with it. A *note string* consists of a *base note* and all of its responses. Separate *notesfiles* contain discussions on separate subject matters; microcomputers might be discussed in a "micronotes" notesfile while bicycling enthusiasts make their comments in a "bicycle" notesfile.

The **–s** option signals *notes* to use the automatic sequencer. With the sequencer enabled, *notes* shows the new notes and responses since your last entry into that notesfile. With the sequencer enabled by **–s** the notes program will not enter notesfiles which have no new text. Specify **–x** to use the sequencer and enter notesfiles even if they have no new text. The **–i** and **–n** options are still more sequencing modes, **–i** is similar to **–s** but shows the index page instead of the first modified note. **–n** turns the sequencer off.

The **-o** option helps users find articles that are vaguely remembered with terms such as "some note in the last 3 days in one of several notesfiles". These can be found with a command like:

```
notes -o "3 days ago" nf1 nf2 nf3
```

which sequences past all articles written in notesfiles "nf1", "nf2", and "nf3" in the last 3 days. The user's timestamps are not updated.

The **-a** option specifies a *subsequencer*. This allows several people sharing the same signon to maintain their own sequencer file. The actual sequencer name is generated by concatenating the user name and the subsequencer name. It is recommended that subsequencer names be unique within the first 6 characters.

Specify **–t** *termtype* to override the TERM environment variable. This switch is primarily for V6 systems.

The **-f** option directs *notes* to read the contents of a file for a list of notesfiles to scan. This file and the directories must be readable by the notesfile user id.

The topic list specifies which notesfiles are to be scanned. The notesfiles are scanned from left to right; upon finishing the first topic, the second is entered. The shell's meta-characters are recognized within a topic but must be escaped to prevent shell interpretation. Specifying "net.*" will yield all the notesfiles with the prefix "net.". Specify "*unix*" to read all notesfiles with the string "unix" in their names. Bracket and question mark constructs are also recognized.

Notesfile names are parsed such that a notesfile will be entered only once no matter how many times it is listed on the command line and in any files specified by the **-f** option. Notesfiles can also be excluded by prefixing their names with a '!'. Thus to see all notesfiles except "general", one might type:

```
notes "*" !general
```

The *autoseq* and *readnotes* commands allow sequencing through a list of notesfiles with the sequencer enabled using with a single command. *Autoseq* and *Readnotes* function identically. They are syntactically equivalent to "notes -s $NFSEQ". The environment variable NFSEQ

contains a comma separated list of notesfile specifications. A typical *NFSEQ* definition for the Bourne shell looks like:

NFSEQ="general,announce,net.*,bicycle,srg,:/usr/essick/nflist".

Specifications beginning with a ':' specify a file to read for more notesfile names. In the previous example, the last specification reads the contents of the file '/usr/essick/nflist' for more notesfile specifications. Many of these can appear in the *NFSEQ* variable.

Notes and responses are entered by using an editor. The default editor is *ed*(1). This can be changed by setting one of the environment variables *NFED* or *EDITOR*. *Notes* looks for *NFED* before looking for *EDITOR*, allowing users to use different editors for writing notes and for other tools.

Some commonly used commands within the notesfile system are listed below:

space Show the next page of the note/response.

; Go the next response, if there are no more responses go to the next note.

- Go to the previous page of the current note/response. From the first page of a response, go to the previous response (or the base note from the first response). From the first page of a base note, go to the previous note.

newline Go to the next note.

j Jump to the next unread note/response (when using sequencer).

J Jump to the next unread note, ignoring any further responses in the current note string (when using sequencer).

w When issued from the index page enters a new note. When entered from a note/response display enters a response. A capital-W will include the text of the currently displayed note/response in the new response.

q Leave the current notesfile.

Q Leave the current notesfile without updating the sequencer information.

control-d Return to the shell, ignoring any further notesfiles in the current invocation. No sequencer information is updated.

x Search for a note with the (prompted for) string in its title. Capital-X asks for a new search string, otherwise the last entered string is used.

s Saves the currently displayed note/response at the end of a (prompted for) file. Capital-S saves the entire note string.

M Sends the text of the note/response displayed and your comments to another user(s). The P command routes the letter to the author of the note/response.

t Issues a write(1) command to the author of the currently displayed note/response. No action is taken if the note originated on a remote system or is anonymous.

! Forks a shell.

/ Searches the text of notes and responses for a regular expression. If an expression isn't specified, the previously specified expression is used. The first page of the note/response that contains the expression is displayed when the expression is found.

Only the *notesfile owner* can create new notesfiles. The *notesfile owner* will create the notesfile and turn control over to the person requesting the notesfile. This person is the *notesfile director;* he may designate others to also be *notesfile directors*. The *notesfile director* has special privileges including: deleting any note, determining policy for the notesfile, permitting anonymous notes, and determining accessibility of the notesfile.

Facilities for mailing to notesfiles (*nfmail*(8)), networking notesfiles (*nfxmit*(8)), printing notesfiles (*nfprint*(1)), archiving old notes (*nfarchive*(8)), and several user routines (*nfabort*(3) and *nfcomment*(3)) exist.

The concept of a notesfile was taken from the PLATO system (a trademark of Control Data Corporation) designed at the University of Illinois to provide automated teaching capabilities.

FILES

/etc/passwd	for the users name
/etc/group	for the users group(s)
/etc/termcap	for terminal capabilities
/usr/spool/notes	the default notesfile data base
/usr/spool/notes/.utilities	utility programs and online help
/usr/spool/notes/.sequencer/*user*	Sequencing timestamps for *user*.
/usr/spool/notes/.sequencer/*user:subsequencer*	Sub-sequencing timestamps for *user*.

SEE ALSO

checknotes(1), ed(1), mknf(8), nfabort(3), nfaccess(8), nfarchive(8), nfmail(8), nfpipe(1), nfprint(1), nfstats(1), nfxmit(8), nfcomment(3), termcap(5), write(1),
The Notesfile Reference Manual in the *ConvexOS Tutorial Papers*

NAME

nroff – text formatting

SYNOPSIS

nroff [option] ... [file] ...

DESCRIPTION

nroff formats text in the named *files* for typewriter-like devices. The full capabilities of *nroff* are described in the *Nroff/Troff User's Manual.*

If no *file* argument is present, the standard input is read. An argument consisting of a single minus (–) is taken to be a file name corresponding to the standard input.

The options, which may appear in any order so long as they appear *before* the files, are:

–o*list* Print only pages whose page numbers appear in the comma-separated *list* of numbers and ranges. A range *N–M* means pages *N* through *M*; an initial *–N* means from the beginning to page *N*; and a final *N–* means from *N* to the end.

–n*N* Number first generated page *N*.

–s*N* Stop every *N* pages. *nroff* will halt prior to every *N* pages (default *N*=**1**) to allow paper loading or changing, and will resume upon receipt of a newline.

–m*name* Prepend the macro file **/usr/lib/tmac/tmac.***name* to the input *files*.

–r*aN* Set register *a* (one-character) to *N*.

–i Read standard input after the input files are exhausted.

–q Invoke the simultaneous input-output mode of the **rd** request.

–T*name* Prepare output for specified terminal. Known *names* include **37** for the (default) Teletype Corporation Model 37 terminal, **tn300** for the GE TermiNet 300 (or any terminal without half-line capability), **300S** for the DASI-300S, **300** for the DASI-300, and **450** for the DASI-450 (Diablo Hyterm). See */usr/lib/term/README* for the complete list of supported terminals.

–e Produce equally-spaced words in adjusted lines, using full terminal resolution.

–h Use output tabs during horizontal spacing to speed output and reduce output character count. Tab settings are assumed to be every 8 nominal character widths.

FILES

/tmp/ta*	temporary file
/usr/lib/tmac/tmac.*	standard macro files
/usr/lib/term/README	complete list of supported terminals
/usr/lib/term/*	terminal driving tables for *nroff*

SEE ALSO

"Nroff/Troff user's manual" in the *ConvexOS Tutorial Papers*
B. W. Kernighan, *A TROFF Tutorial*
neqn(1), tbl(1), ms(7), me(7), man(7), col(1)

NAME

nslookup – query name servers interactively

SYNOPSIS

nslookup [*host-to-find* | – [*server address* | *server name*]]

DESCRIPTION

nslookup is a program to query DARPA Internet domain name servers. nslookup has two modes: interactive and non-interactive. Interactive mode allows the user to query the name server for information about various hosts and domains or print a list of hosts in the domain. Non-interactive mode is used to print just the name and Internet address of a host or domain.

ARGUMENTS

Interactive mode is entered in the following cases:

a) When no arguments are given (the default name server will be used).

b) When the first argument is a hyphen (–) and the second argument is the host name of a name server.

Non-interactive mode is used when the name of the host to be looked up is given as the first argument. The optional second argument specifies a name server.

INTERACTIVE COMMANDS

Commands may be interrupted at any time by typing a control-C. To exit, type a control-D (EOF). The command line length must be less than 80 characters. An unrecognized command will be interpreted as a host name.

host [*server*] Look up information for *host* using the current default server or using *server* if it is specified.

server *domain*
lserver *domain*
Change the default server to *domain*. **lserver** uses the initial server to look up information about *domain* while **server** uses the current default server. If an authoritative answer can't be found, the names of servers that might have the answer are returned.

root Changes the default server to the server for the root of the domain name space. Currently, the host sri-nic.arpa is used. (This command is a synonym for the **lserver sri-nic.arpa**.) The name of the root server can be changed with the **set root** command.

finger [*name*] [> *filename*]
finger [*name*] [>> *filename*]
Connects with the finger server on the current host. The current host is defined when a previous lookup for a host was successful and returned address information (see the **set querytype=A** command). *Name* is optional. > and >> can be used to redirect output in the usual manner.

ls *domain* [> *filename*]
ls *domain* [>> *filename*]
ls -a *domain* [> *filename*]
ls -a *domain* [>> *filename*]
ls -h *domain* [> *filename*]
ls -h *domain* [>> *filename*]

ls -d *domain* [> *filename*]
List the information available for *domain*. The default output contains host names and their Internet addresses. The **-a** option lists aliases of hosts in the domain. The **-h** option lists CPU and operating system information for the domain. The **-d** option lists all contents of a zone transfer. When output is directed to a file, hash marks are printed for every 50 records received from the server.

view *filename*
Sorts and lists the output of previous **ls** command(s) with *more*(1).

help

? Prints a brief summary of commands.

set *keyword*[=*value*]
This command is used to change state information that affects the lookups. Valid keywords are:

all Prints the current values of the various options to **set**. Information about the current default server and host is also printed.

[no]debug
Turn debugging mode on. A lot more information is printed about the packet sent to the server and the resulting answer.
(Default = nodebug, abbreviation = [no]deb)

[no]d2 Turn exhaustive debugging mode on. Essentially all fields of every packet are printed.
(Default = nod2)

[no]defname
Append the default domain name to every lookup.
(Default = defname, abbreviation = [no]def)

[no]search
With defname, search for each name in parent domains of the current domain.
(Default = search)

domain=*name*
Change the default domain name to *name*. The default domain name is appended to all lookup requests if the **defname** option has been set. The search list is set to parents of the domain with at least two components in their names.
(Default = value in hostname or /etc/resolv.conf, abbreviation = do)

querytype=*value*

type=*value*
Change the type of information returned from a query to one of:

A	the host's Internet address (the default).
CNAME	the canonical name for an alias.
HINFO	the host CPU and operating system type.
MD	the mail destination.
MX	the mail exchanger.
MG	the mail group member.
MINFO	the mailbox or mail list information.

MR the mail rename domain name.

NS nameserver for the named zone.

Other types specified in the RFC883 document are valid but aren't very useful. (Abbreviation = q)

[no]recurse
Tell the name server to query other servers if it does not have the information. (Default = recurse, abbreviation = [no]rec)

retry=*number*
Set the number of retries to *number*. When a reply to a request is not received within a certain amount of time (changed with **set timeout**), the request is resent. The retry value controls how many times a request is resent before giving up.
(Default = 2, abbreviation = ret)

root=*host*
Change the name of the root server to *host*. This affects the **root** command.
(Default = sri-nic.arpa, abbreviation = ro)

timeout=*number*
Change the time-out interval for waiting for a reply to *number* seconds.
(Default = 10 seconds, abbreviation = t)

[no]vc Always use a virtual circuit when sending requests to the server.
(Default = novc, abbreviation = [no]v)

DIAGNOSTICS

If the lookup request was not successful, an error message is printed. Possible errors are:

Time-out
The server did not respond to a request after a certain amount of time (changed with **set timeout=***value*) and a certain number of retries (changed with **set retry=***value*).

No information
Depending on the query type set with the **set querytype** command, no information about the host was available, though the host name is valid.

Non-existent domain
The host or domain name does not exist.

Connection refused
Network is unreachable
The connection to the name or finger server could not be made at the current time. This error commonly occurs with **finger** requests.

Server failure
The name server found an internal inconsistency in its database and could not return a valid answer.

Refused
The name server refused to service the request.

The following error should not occur and it indicates a bug in the program.

Format error
The name server found that the request packet was not in the proper format.

FILES

/etc/resolv.conf initial domain name and name server addresses.

SEE ALSO

resolver(3), resolver(5), named(8), RFC882, RFC883

AUTHOR

Andrew Cherenson

NAME

od – octal, decimal, hex, ascii dump

SYNOPSIS

od [–format] [file] [[+]offset[.][**b**] [label]]

DESCRIPTION

Od displays *file*, or its standard input, in one or more dump formats as selected by the first argument. If the first argument is missing, **–o** is the default. Dumping continues until end-of-file.

The meanings of the format argument characters are:

a Interpret bytes as characters and display them with their ACSII names. If the **p** character is given also, then bytes with even parity are underlined. The **P** character causes bytes with odd parity to be underlined. Otherwise the parity bit is ignored.

b Interpret bytes as unsigned octal.

c Interpret bytes as ASCII characters. Certain non-graphic characters appear as C escapes: null=\0, backspace=\b, formfeed=\f, newline=\n, return=\r, tab=\t; others appear as 3-digit octal numbers. Bytes with the parity bit set are displayed in octal.

d Interpret (short) words as unsigned decimal.

e Interpret long words as IEEE mode floating point. This mode is only available on machines with IEEE hardware support; otherwise, *od* will abort with an error message.

f Interpret long words as floating point. The floating point mode used (IEEE or native) will be determined using the *getsysinfo()* system call to retrieve the site default mode.

h Interpret (short) words as unsigned hexadecimal.

i Interpret (short) words as signed decimal.

l Interpret long words as signed decimal.

n Interpret long words as native mode floating point.

o Interpret (short) words as unsigned octal.

s[n] Look for strings of ascii graphic characters, terminated with a null byte. *N* specifies the minimum length string to be recognized. By default, the minimum length is 3 characters.

v Show all data. By default, display lines that are identical to the last line shown are not output, but are indicated with an "*" in column 1.

w[n] Specifies the number of input bytes to be interpreted and displayed on each output line. If **w** is not specified, 16 bytes are read for each display line. If *n* is not specified, it defaults to 32.

x Interpret (short) words as hexadecimal.

An upper case format character implies the long or double precision form of the object.

The *offset* argument specifies the byte offset into the file where dumping is to commence. By default this argument is interpreted in octal. A different radix can be specified; If "**.**" is appended to the argument, then *offset* is interpreted in decimal. If *offset* begins with "**x**" or "**0x**", it is interpreted in hexadecimal. If "**b**" ("**B**") is appended, the offset is interpreted as a block count, where a block is 512 (1024) bytes. If the *file* argument is omitted, an *offset* argument must be preceded by "+".

The radix of the displayed address will be the same as the radix of the *offset*, if specified; otherwise it will be octal.

Label will be interpreted as a pseudo-address for the first byte displayed. It will be shown in "()" following the file offset. It is intended to be used with core images to indicate the real memory address. The syntax for *label* is identical to that for *offset*.

SEE ALSO

adb(1)

BUGS

A file name argument can't start with "+". A hexadecimal offset can't be a block count. Only one file name argument can be given.

It is an historical botch to require specification of object, radix, and sign representation in a single character argument.

NAME

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

SYNOPSIS

oldcsh [**–cefinstvVxX**] [arg ...]

DESCRIPTION

oldcsh 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. To be able to use its job control facilities, users of *oldcsh* must (and automatically) use the new tty driver fully described in *tty*(4). This new tty driver allows generation of interrupt characters from the keyboard to tell jobs to stop. See *stty*(1) for details on setting options in the new tty driver.

An instance of *oldcsh* begins by executing commands from the file *.cshrc* in the *home* directory of the invoker. Typical items to include in a *.cshrc* file are aliases and other variable settings. There are certain advantages if you avoid setting a lot of aliases or your prompt when the current shell is not interactive. This can be accomplished by the inclusion of the line

if (! $?prompt) exit

in your *.cshrc* file immediately after the lines that you want to have executed *every* time a shell is invoked. If this is a login shell, then *oldcsh* also executes commands from the global login file */etc/login*, and from the home directory file *.login*, in that order. It is typical for users on crt's to put the command **stty crt** in their *.login* file and to invoke *tset*(1) there.

In the normal case, the shell then begins 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 first from the file *.logout* in the users home directory and then from the global logout file */etc/logout*.

Lexical structure

The shell splits input lines into words at blanks and tabs with the following exceptions. The characters &, |, ;, <, >, (, and) form separate words. If doubled in &&, ||, <<, or >>, these pairs form single words. These parser metacharacters may be made part of other words. To prevent their special meaning, precede 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 that continues to the end of the input line. This special meaning is prevented when the # is preceded by \ and is 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 that 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 that 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 keys ^Z (control-Z). This sends a STOP signal to the current job. The shell normally indicates that the job has been "Stopped" and prints 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 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. There is another special command, ^Y, which does not generate a STOP signal until a program attempts to *read*(2) it. This can usefully be typed ahead when you have prepared some commands for a job that you wish to stop after it has read them.

A job being run in the background stops 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, background jobs will stop when they try to produce output, just as they do when they try to read input.

There are several ways to refer to jobs in the shell. The character % introduces a job name. If you wish to refer to job number 1, you can name it as **%1**. Just naming a job brings it to the foreground; thus **%1** is a synonym for **fg %1**, bringing job 1 back into the foreground. Similarly, entering **%1 &** resumes job 1 in 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**. This 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 notifies you immediately of changes of status in background jobs. There is also a shell command *notify* that marks a single process so that its status changes are immediately reported. By default, *notify* marks the current process; simply say **notify** after starting a background job to mark it.

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

Substitutions

The various transformations the shell performs on the input are described in the sections below 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 that contains history substitution is echoed on the terminal before it is executed as it could have been typed without history substitution.

Commands that are input from the terminal and that 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. Commands are numbered sequentially from 1. The size of the history list is controlled by the *history* variable. The previous command is always retained, regardless of its value.

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

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

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

With the current event (13) we can refer to previous events by event number **!11**, relatively (as in **!–2**, referring to the same event), by a prefix of a command word (as in **!d** for event 12, or **!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, follow the event specification by **:** and a designator for the desired words. The words of an 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 ↑, $, *, –, or %. A sequence of modifiers, each preceded by **:**, can be placed after the optional word designator. 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 *r* for *l*
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., **!$**. In this case, the reference is to the previous command. However, if a previous history reference occurred on the same line, this form repeats the previous reference. Thus **!?foo?↑ !$** gives the first and last arguments from the command matching **?foo?**.

The special history reference **!#** does not refer to any previous command, but refers to the current command. It can be used to duplicate any or all of the words just typed. For example, **ls /usr/src/lib !#^/libc** is equivalent to **ls /usr/src/lib /usr/src/lib/libc**.

A special abbreviation of a history reference occurs when the first nonblank character of an input line is a ↑. This is equivalent to **!:s↑** providing a convenient shorthand for substitutions on the text of the previous line. Thus, **↑lb↑lib** fixes the spelling of **lib** in the previous command.

Finally, a history substitution may be surrounded with **{** and **}**, if necessary, to insulate it from the characters that follow. Thus, after **ls –ld ~paul**, you might enter **!{l}a** to do **ls –ld ~paula**, while **!la** would look for a command starting with **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 that can be established, displayed, and modified by the *alias* and *unalias* commands. After a command line is scanned, it is parsed into distinct commands, and the first word of each command, left-to-right, is checked to see if it is an alias. If it is, the alias text is reread with the history mechanism available as though that command were the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, then the argument list is left unchanged.

Thus, if the alias for **ls –l** is **ls**, the command **ls /usr** would map to **ls –l /usr**, the argument list here being undisturbed. Similarly, if **lookup** was an alias for **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 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 that *pr*'s its arguments to the line printer.

Variable substitution

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

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

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

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

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

Unless enclosed in " or given the **:q** modifier, the results of variable substitution may eventually be command and filename substituted. Within ", a variable whose value consists of multiple words expands to a portion of a single word, with the words of the variables value separated by blanks. When the **:q** modifier is applied to a substitution, the variable expands 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 that 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 that 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 (but **:** modifiers and the other forms given below are not available in this case).

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

$#name
${#name}
Gives the number of words in the variable. This is useful for later use in a [selector].

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

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

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

The modifiers **:h**, **:t**, **:r**, **:q**, and **:x** may be applied to the substitutions above, as may **:gh**, **:gt**, and **:gr**. If braces { } appear in the command form, then the modifiers must appear within the braces. *The current implementation allows only one* **:** *modifier on each* $ *expansion.*

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

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

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

$$
Substitutes 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 substitutions, are applied selectively to the arguments of builtin commands. This means that portions of expressions that are not evaluated are not subjected to these expansions. For commands that 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 single quotation marks. 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 souble quotation marks, only newlines force new words; blanks and tabs are preserved.

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

Filename substitution

If a word contains any of the characters *, ?, [, {, or 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 filenames that match the pattern. In a list of words specifying filename substitution, it is an error for no pattern to match an existing filename, but it is not required for each pattern to match. Only the metacharacters *, ?, and [imply pattern matching, the characters ~ and { are more akin to abbreviations.

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

The ~ at the beginning of a filename is used to refer to home directories. Standing alone i.e., ~, it expands to the invoker's 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 the user's home directory. Thus, **~ken** might expand to **/usr/ken** and **~ken/chmach** to **/usr/ken/chmach**. If the ~ is followed by a character other than a letter or /, or appears other than at the beginning of a word, it is left undisturbed.

The metanotation **a{b,c,d}e** is 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 that is identical to *word*. *word* is not subjected to variable, filename, or command substitution, and each input line is compared to *word* before any substitutions are done on this input line. Unless a quoting \, ", ', or ` appears in *word*, variable and command substitution is performed on the intervening lines, allowing \ to quote $, \, and `. Commands that 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 that 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, it is truncated and its previous contents are lost.

  If the variable *noclobber* is set, then the file must not exist or be a character special file (e.g., a terminal or **/dev/null**) or an error results. This helps prevent accidental destruction of files. In this case, the **!** forms can be used 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 that < 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 that run from a file of shell commands have no access to the text of the commands by default; rather, they receive the original standard input of the shell. The << mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block-read its input. Note that the default standard input for a command run detached 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 =~ is pattern equivalence and !~ is pattern non-equivalence much as != is numerical non-equivalence and == is numerical equivalence. Pattern (non)equivalence means the right side of the expression is a *pattern* (containing, e.g., *, ?, and instances of [...]) against which the left 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 that begin with **0** are considered octal numbers. Null or missing arguments are considered **0**. The result of all expressions are strings that 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 that are syntactically significant to the parser (&, |, <, >, (, and)), 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

l symbolic link
p named pipe (fifo)

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

Note that the **–x** test checks only the permission bits on the file in question (see *access (2)*. As a result, this test can be fooled into giving false information, especially in the cases of set-user/group-ID binaries in non-SUID NFS filesystems, set-user/group-Id shell scripts, and generic ASCII files with their execute bits set.

Control flow

The shell contains a number of commands that 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 shells 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 gotos will succeed on nonseekable inputs.)

Builtin commands

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

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

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

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 shell's 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 remaining 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. See *tset*(1) for an example of using *eval*.

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 this command is read from the terminal, the loop is read up once prompting with **?** before any statements in the loop are executed. If you make a mistake entering in a loop at the terminal, you can rub it out.

glob *wordlist*

glob is like *echo*, but no \ escapes are recognized and words are delimited by null characters in the output. This is useful for programs that wish to use the shell to filename expand a list of words.

goto word

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

hashstat

Prints a statistics line indicating how effective the internal hash table has been at locating commands and avoiding *exec*s. An *exec* is attempted for each component of the *path*, where the hash function indicates a possible hit, and in each component that does not begin with a /.

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. 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 remaining *if* command. *Command* must be a simple command, not a pipeline, a command list, or a parenthesized command list. Input/output redirection occurs even if *expr* is false, when command is *not* executed (this is a bug).

if *(expr)* **then**
...
else if *(expr2)* **then**
...
else
...
endif

If the specified *expr* is true, then the commands to the first *else* are executed; **else if** *expr2* is true then the commands to the second else are executed, etc. Any number of 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, entering **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 is sent a CONT (continue) signal as well.

The terminate signal kills processes that do not catch the signal; **kill –9 ...** is a sure kill, as the KILL (9) signal cannot be caught. The killed process must belong to the current user unless the user is the super user.

limit
limit *resource*
limit *resource maximum-use*
Limits the consumption by the current process and each process it creates to not individually exceed *maximum-use* on the specified *resource*. If no *maximum-use* is given, then the current limit is printed; if no *resource* is given, then all limitations are given.

Resources that are controllable currently include *cputime* (the maximum number of cpu-seconds to be used by each process), *filesize* (the largest single file that can be created), *datasize* (the maximum growth of the data+stack region via *sbrk*(2) beyond the end of the program text), *stacksize* (the maximum size of the automatically-extended stack region), *coredumpsize* (the size of the largest core dump that will be created), *memoryuse* (the maximum size a processs resident set size may grow to), and *concurrency* (the maximum number of threads a process is allowed to have running in parallel).

The *maximum-use* may be given as a floating-point or integer number followed by a scale factor. For all limits other than *cputime* and *concurrency*, the default scale is **k** or **kilobytes** (1024 bytes); a scale factor of **m** or **megabytes** may also be used. For *cputime*, the default scaling is **seconds**, while **m** for minutes or **h** for hours, or a time of the form *mm:ss* giving minutes and seconds may be used. There is no need for a scale factor for *concurrency*, since the values will be integers between one and the number of processors on the system, inclusively.

For both *resource* names and scale factors, unambiguous prefixes of the names suffice.

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 superuser may specify negative niceness by using **nice**-number .... command is always

executed in a subshell; *command* must be a simple command, not a pipeline, an alias, a command list, or a parenthesized command.

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 that are to terminate shell scripts or to return to the terminal command input level. The second form **onintr** – causes all interrupts to be ignored. The final form causes the shell to execute a **goto label** when an interrupt is received or a child process terminates because it was interrupted.

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

popd
popd *+n*
Pops the directory stack, returning to the new top directory. With an 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. Given a *name* argument, *pushd* pushes the current working directory (as in *cwd*) onto the directory stack, then changes to the new directory (ala *cd*). With a numeric argument, 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 value print as a parenthesized word list. The second form sets *name* to the null string. The third form sets *name* to the single *word*. The fourth form sets the *index*th component of name to word; this component must already exist. The final form sets *name* to the list of words in *wordlist*. In all cases, the value is command and filename expanded.

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

setenv *name value*

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 *oldcsh* 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. Note that since *source* is a *oldcsh* builtin function, ^Z and ^C will not work in it. Attempting to stop or suspend the job will result in an attempt to stop or suspend the shell.

stop
stop *%job ...*

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

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

umask
umask *value*
The file creation mask is displayed (first form) or set to the specified value (second form). The mask is given in octal. Common values for the mask are 002, giving all access to the group and read and execute access to others, or 022, giving yourself all access, but 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 if nothing is *unaliased.*

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

unlimit *resource*
unlimit
Removes the limitation on *resource.* If no *resource* is specified, then all *resource* limitations are removed.

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 names match the specified pattern from the environment. See 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.

whence *name ...*
whence –w *name ...*
Searches aliases, builtins, and finally the user's *path* for the program file that would be executed if *name* were to be typed as a command. The output can be in one of three forms: the full path of the command that would be executed, an indication of the wordlist that *name* is aliased to, or notification that the command in question is built in to the shell. If no output is given, the command was not found anywhere in the user's *path,* as a C shell builtin, or as an alias. *whence* may be invoked with multiple names, and each one will be processed in order. If the second form is used, the output will be in the same style as *which*(1). The first form is perhaps more complete since it reports on C shell builtin commands, but both forms run much faster than the *which*(1) utility.

Examples:

```
% whence which
/usr/ucb/which
% alias h history
% whence h
```

```
alias/h 'history'
% whence history
builtin/history
```

while *(expr)*
...
end

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

%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 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, however, are mandatory in separating components of *expr* that would otherwise be single words.

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

Predefined and environment variables

The following variables have special meaning to the shell. Of these, *argv, autologout, 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 are not 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 likewise handled. It is not necessary to worry about its setting other than in the file *.cshrc,* as inferior *oldcsh* processes import the definition of *path* from the environment and re-export it if you 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.

autologout Set to the number of minutes an interactive shell can be idle before the shell automatically terminates itself. The default value for *autologout* is 0; *autologout* disabled. For the shell, idle time is defined to be the time between when the *prompt* is printed and when a command is entered. The *autologout* feature is disabled by setting its value to 0.

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 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 that has been referenced in this many events is not 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 that are terminals. This prevents shells from accidentally being killed by **CTRL-D**'s.

mail The files where the shell checks for mail. This is done after each command completion that results 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 ensure that files are not accidentally destroyed and that **>>** redirections refer to existing files.

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

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

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 pathnames will execute. The usual search path is ., */bin,* and */usr/bin,* but this may vary from system to system. For the superuser, the default search path is */etc, /bin,* and */usr/bin.* A shell which is given neither the **–c** nor the **–t** option normally hashes 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* builtin, or the commands may not be found.

prompt The string that is printed before each command is read from an interactive terminal input. If a **!** appears in the string, it is replaced by the current event number unless a preceding **** is given. Default is **%**, or **#** for the superuser.

savehist 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 that 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. Values of *savehist* that are too large 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 that have execute bits set, but that are not executable by the system. (See the description of **Non-builtin Command Execution** below.) *Shell* is 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 that fail return exit status **1**; all other builtin commands set status **0**.

time The first word of this array variable is numeric and controls automatic timing of commands. If set, then any command that takes more than this many CPU seconds causes a line to be printed that gives a time and resource usage summary (most of the data is from *getrusage*(2)). The second word of this variable, which takes the form of a *printf*-like format string, can be used to customize the output of the *time* builtin. The following conversion specifications are recognized and have the given meanings:

- %U Amount of time spent executing in user mode (in seconds).
- %S Amount of time spent in the system executing on behalf of the process (in seconds).
- %E Real (elapsed) time in minutes and seconds.
- %P Percentage of CPU utilization (which is the ratio of user plus system times to real time, scaled by the number of processors on the system).
- %C CPU parallelization factor or concurrency level (somewhere between one and the number of processors on the system). This value is designed to be used when timing a particular process and so will display *N/A* when *time* is issued as a command with no arguments.
- %W Number of process swaps out of main memory.
- %X Amount of memory shared among other processes (in kilobytes).
- %D Combined size of unshared data and stack segments (in kilobytes).
- %K Total size of shared memory, unshared data, and unshared stack sizes.
- %M Maximum resident set size utilized (in kilobytes).
- %F Number of page faults serviced which required I/O activity.
- %R Number of page faults serviced without I/O activity; here, I/O activity is avoided by reclaiming a page from the list of pages awaiting reallocation.
- %I Number of times the file system had to perform block input.
- %O Number of times the file system had to perform block output.

The default format string for *time* output is

%Uu %Ss %E %P %X+%Dk %I+%Oio %Fpf+%Ww,

which will print user, system, and real times; a utilization percentage; the amount of shared memory; the combined size of unshared data and stack

segments; the number of times the file system had to perform block input and output; the number of page faults; and the number of process swaps.

Examples:

```
% set time = 0
% date
Sat Jun 21 16:30:00 CDT 1986
0.00u 0.01s 0:00 60% 0+0k 1+0io 10pf+0w
% set time = (0 "user=%U sys=%S real=%E; %C‖")
% date
Sat Jun 21 16:30:07 CDT 1986
user=0.00 system=0.01 real=0:00; 1.3‖
```

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

Non-builtin command execution

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

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

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

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

Argument list processing

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

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

–e The shell exits if any invoked command terminates abnormally or yields a nonzero exit status.

–f The shell starts faster because it neither searches for nor executes commands from the file *.cshrc* in the invoker's home directory.

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

-n Commands are parsed, but not executed. This 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**.

After processing of flag arguments, if 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 **&**, **bg**, or **%... &** commands) are immune to signals generated from the keyboard, including hangups. Other signals have the values that the shell inherited from its parent. The shells handling of 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 shells parent. In no case are interrupts allowed when a login shell is reading the file *.logout*.

FILES

~/.cshrc	Read at beginning of execution by each shell.
~/.login	Read by login shell after */etc/login* 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/login	Read by login shell after *.cshrc* at login.
/etc/logout	Read by login shell after *.logout* at logout.
/etc/passwd	Source of home directories for **~name**.

LIMITATIONS

Words can be no longer than 1024 characters. The system limits argument lists to 10240 characters. The number of arguments to a command, which involves filename expansion, is limited to 1/6 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), killpg(2), pipe(2), sigvec(2), umask(2), getrlimit(2), wait(2), tty(4), a.out(5), environ(7)

"An Introduction to the C Shell" in the *ConvexOS Tutorial Papers*

BUGS

When a command is restarted from a stop, the shell prints the directory it started in, if this is different from the current directory. This can be misleading (i.e., wrong) because the job may have changed directories internally.

Shell builtin functions are not stoppable or restartable. Command sequences of the form **a ; b ; c** are also not handled gracefully when stopping is attempted. If you suspend **b**, the shell immediately executes **c**. This is especially noticeable if this expansion results from an *alias*. It suffices to place the sequence of commands in ( ) 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. In a virtual terminal interface, much more interesting 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 builtin 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.

Symbolic links fool the shell. In particular, **dirs** and **cd ..** do not work properly once you have crossed through a symbolic link.

The **–x** test for executability can be fooled (see the **Expressions** section for more information).

NAME

opreq – Operator Request handler

SYNOPSIS

opreq

DESCRIPTION

opreq is a program that handles requests to an operator. Request are currently limited to such things as tape mount requests. Within *opreq* requests are generally referred to as "messages".

By default, *opreq* is started with one window. This window can be closed and new ones opened if desired. Each window can be configured to display only certain types of messages and only messages with a certain status. The window title, which lists the columns of the message to display, can also be configured (see the configure commands below.)

If *opreq* can read a file called .opreqrc in either the current working directory or the home directory of the user, this file is opened and read for the initial configuration of windows rather than the system wide .opreqrc file in /usr/lib/opreq.

COMMANDS

Commands are entered by typing characters at the "Command: " prompt. For each character typed, *opreq* will fill in the rest of the command line from the list of available commands. Once a complete command is filled in, RETURN causes the command to be executed. For example, the following keystrokes would be used execute the *field-vsn* command:

1. The user presses "f" and *opreq* fills in "field-".
2. The user presses "v" and *opreq* fills in "vsn".
3. The user presses RETURN and *opreq* executes the *field-vsn* command.

Pressing the question mark (?) key at any time will give you a list of possible commands. Typing a number before some commands will repeat that command that many times. For commands where a number doesn't make sense, the number is ignored.

The following is a list of the top-level commands available in *opreq*. Some commands have set of subcommands available, such as *configure-status* or *window-goto*.

configure-drives This command allows members of the *operator* group to enable and disable drives in *tpdaemon*. The *configure-drives* command creates a small pop-up window with a list of known drives. Drives marked with an "X" are drives that will be controlled by *tpdaemon*. Drives not marked with an "X" are drives that will not be controlled by *tpdaemon*. This control state can be toggled by pressing RETURN while a particular drive is highlighted. Typing **j** or CTRL-N will highlight the next selection; typing **k** or CTRL-P will highlight the previous selection. Entering *cancel* will cancel the command. Entering *done* causes the selected drives to be controlled by tpdaemon and the drives not selected to be not controlled by tpdaemon.

configure-status This command controls what status a message must be to be displayed in the window. After entering the *configure-status* command, a small, pop-up window will open with a list of possible statuses. Each status will show its current state, either off or on. An "X" will appear next to the currently selected statuses. This state can be toggled by pressing RETURN while that status is highlighted. Typing **j** or CTRL-N will highlight the next selection; typing **k** or CTRL-P will highlight the previous selection. Entering *cancel* will cancel the command. Entering *done* causes the selected message statuses to be displayed in the current window.

configure-title This command controls what columns of a message are displayed in the window. After typing the *configure-title* command, a small, pop-up window will open with a list of possible columns. Each column will show its current state, either off or on. An "X" will appear next to the currently selected columns. This state can be toggled by pressing RETURN while that status is highlighted. Typing **j** or CTRL-N will highlight the next selection; typing **k** or CTRL-P will highlight the previous selection. Entering *cancel* will cancel the command. Entering *done* causes the selected columns to be displayed in the

current window.

configure-type This command controls what types of messages are displayed in the window. After typing the *configure-type* command, a small, pop-up window will open with a list of possible types. Each type will show its current state, either off or on. An "X" will appear next to the currently selected types. This state can be toggled by pressing **RETURN** while that status is highlighted. Typing **j** or **CTRL-N** will highlight the next selection; typing **k** or **CTRL-P** will highlight the previous selection. Entering *cancel* will cancel the command. Entering *done* causes the selected message types to be displayed in the current window.

display-available-drives
Displays a list of currently available drives.

display-messages Displays, or redisplays, the list of current messages. It always starts at the beginning of the messages list. This will remove messages whose status or type no longer match that of the window.

field-assigned Displays the "assigned" field of the current message. A message is assigned to the user that has executed *select-work* on a message. If the message is assigned to someone, the user ID of that person will be displayed. Otherwise, it will be blank.

field-comment Displays the "comment" field of the current message.

field-drive Displays the "drive" field of the current message. This will show the drive associated with the current message.

field-link Displays the "link" field of the current message. The link field contains the message ID (mid) of the message it is linked to, zero means it is not linked to another message.

field-mid Displays the "mid" field of the current message. The mid is a unique number that was assigned to the message.

field-ring Displays the "ring" field of the current message. The ring field is specific to tape requests. It tells whether or not a tape should have read/write or read only protection.

field-status Displays the "status" field of the current message.

field-time-in Displays the "time-in" field of the current message. The is the time that the message arrived into the system.

field-time-out Displays the "time-out" field of the current message. The is the time that the message is resolved. This field will only be valid after the message is resolved with a status of *done* or *canceled.*

field-type Displays the "type" field of the current message. The type can be any of the following: *Mount-Tape, Unmount-Tape, Auto-Vol-Rec, Replace-Tape, Problem, Info, Cancel,* and *Silo-Enter*

field-uid Displays the "uid" field of the current message. This is the user ID of the user who issued the request.

field-vsn Displays the "vsn" field of the current message. If a request has multiple Volume Serial Numbers, or VSNs, associated with it, a window pops up with all the VSNs for that request. The VSN that needs immediate attention is highlighted.

move-down, **j**, or **CTRL-N**
Causes the next message to be highlighted; scrolling if necessary. The down arrow key does the same thing.

move-up, **k**, or **CTRL-P**
Causes the previous message to be highlighted; scrolling if necessary. The up arrow key does the same thing.

redraw or **CTRL-L**
Redraws the screen.

select-cancel Cancels the current message. You must be running as root or be a member of the operator group to perform this command.

select-done Changes the status of the current message to *done*. If the message is a mount request, it will first open a window with a list of drives and prompt you to pick one (by pressing return) to indicate where you placed the tape. An "X" will appear next to the currently available drives. "cancel" will cancel the command. You must be running as root or be a member of the operator group to perform this command.

select-work Selects the current message for working. This should be used to prevent two people from attempting to work on the same message at the same time. You must running as Super User or be a member of the *operator* group to perform this command.

update-display Updates the current window. This will add new messages that have arrived since the last update-display or display-message command was done. This command is done every 10 seconds automatically.

window-close Closes the current window.

window-goto Selects another window to be highlighted. After typing this command a cursor will be put on the screen. Simply use the *move-up*, *move-down*, *move-left*, and *move-right* commands to put the cursor in the desired window (arrow keys also work here.) *put-window* or RETURN will choose the window you are in. *cancel* will cancel the command. Also, a specific window can be selected by entering the number of the desired window before the command. If there are only two windows, it will automatically go to the other window.

window-open Opens a new window. After typing this command a small box will be put on the screen. Simply use the *move-up*, *move-down*, *move-left*, and *move-right* commands to position the upper left corner of the new window (arrow keys also work here.) *put-window* or RETURN puts the upper left corner of the window in the current position. Next, use the *move* commands to position the bottom right corner. *select* or RETURN puts the bottom right corner of the window in the current position. *cancel* will cancel the command.

window-resize Not yet implemented.

quit Quits the session of *opreq*.

FILES

/usr/lib/opreq/share.mem	Shared memory file
/usr/lib/opreq/opreq_share.lockfile	Shared memory lock file
/usr/lib/opreq/opreq_daemon.lockfile	opreq_daemon lock file
/usr/lib/opreq/.opreqrc	System opreq start up file
~/.opreqrc	User's opreq start up file

SEE ALSO

opreq_daemon(8), tpconfig(8), tpdaemon(8)

NAME

pagesize – print system page size

SYNOPSIS

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.

SEE ALSO

getpagesize(2)

NAME

passwd – change login password

SYNOPSIS

passwd [**–f** *filename*] [*username*]

DESCRIPTION

This command changes (or installs) a password associated with the *username* (your own name by default). If the **-f** option is given, *filename* is treated as the password file.

The program prompts for the old password and then for the new one. You must supply both, and the new password must be typed twice to forestall mistakes.

The first time the new password is entered, *passwd* checks to see if any password restrictions are enabled for *username*. Password aging is enabled if there is a non-null age field for *username* in the restrictions file */etc/pwrestrict*; password typing is enabled if a "Y" appears in the type field of the user's entry in the restrictions file.

If "aging" is enabled, and the password has not "aged" sufficiently (the minimum number of weeks a password must remain unchanged has not elapsed), the new password is rejected and *passwd* terminates (see *passwd*(5)). If password typing is enabled, a check is made to insure that the new password meets construction requirements. These requirements are:

Each password must have at least six characters.

Each password must contain at least two alphabetic characters and at least one numeric or special character. In this case "alphabetic" means upper and lower case letters.

Each password must differ from the user's *login* name and any reverse or circular shift of that name. For comparison purposes, an upper case letter and its corresponding lower case letter are equivalent.

New passwords must differ from the old by at least three characters. As before, an upper case letter and its lower case counterpart are equivalent.

Even if no type restrictions are enabled, new passwords must be at least four characters in length if they combine upper-case, lower-case, and numeric characters; at least five characters long if they combine characters from two of the upper-case, lower-case, and numeric character sets; or at least six characters long if monocase.

Only the owner of the name or the superuser may change a password; the owner must prove he knows the old password.

Use *yppasswd* to change your password in the network yellow pages. This will not affect your local password, or your local password on any remote machines on which you have accounts.

FILES

/etc/passwd
/etc/passwd.dir
/etc/passwd.pag
/etc/yp/passwd
/etc/pwrestrict
/etc/pwrestrict.dir
/etc/pwrestrict.pag
/etc/yp/pwrestrict

SEE ALSO

login(1), passwd(5), pwrestrict(5), crypt(3), yppasswd(1), chsh(1), chfn(1), mkpasswd(8)

RESTRICTIONS

Passwd will accept passwords longer than eight characters, but only the first eight characters will be used.

BUGS

Passwd will change a local password, but not a password in the network Yellow Pages. Refer to *yppasswd*(1) for information on how to change a Yellow Pages password.

Since *passwd* rebuilds the passwd and pwrestrict database files, a large password file may produce a long wait.

NAME

patch - a program for applying a diff file to an original

SYNOPSIS

patch [options] orig patchfile [+ [options] orig]

but usually just

patch <patchfile

DESCRIPTION

Patch will take a patch file containing any of the three forms of difference listing produced by the *diff* program and apply those differences to an original file, producing a patched version. By default, the patched version is put in place of the original, with the original file backed up to the same name with the extension ".orig", or as specified by the **-b** switch. You may also specify where you want the output to go with a **-o** switch. If *patchfile* is omitted, or is a hyphen, the patch will be read from standard input.

Upon startup, patch will attempt to determine the type of the diff listing, unless over-ruled by a **-c**, **-e**, or **-n** switch. Context diffs and normal diffs are applied by the *patch* program itself, while ed diffs are simply fed to the *ed* editor via a pipe.

Patch will try to skip any leading garbage, apply the diff, and then skip any trailing garbage. Thus you could feed an article or message containing a diff listing to *patch*, and it should work. If the entire diff is indented by a consistent amount, this will be taken into account.

With context diffs, and to a lesser extent with normal diffs, *patch* can detect when the line numbers mentioned in the patch are incorrect, and will attempt to find the correct place to apply each hunk of the patch. As a first guess, it takes the line number mentioned for the hunk, plus or minus any offset used in applying the previous hunk. If that is not the correct place, *patch* will scan both forwards and backwards for a set of lines matching the context given in the hunk. First *patch* looks for a place where all lines of the context match. If no such place is found, and it's a context diff, and the maximum fuzz factor is set to 1 or more, then another scan takes place ignoring the first and last line of context. If that fails, and the maximum fuzz factor is set to 2 or more, the first two and last two lines of context are ignored, and another scan is made. (The default maximum fuzz factor is 2.) If *patch* cannot find a place to install that hunk of the patch, it will put the hunk out to a reject file, which normally is the name of the output file plus ".rej". (Note that the rejected hunk will come out in context diff form whether the input patch was a context diff or a normal diff. If the input was a normal diff, many of the contexts will simply be null.) The line numbers on the hunks in the reject file may be different than in the patch file: they reflect the approximate location patch thinks the failed hunks belong in the new file rather than the old one.

As each hunk is completed, you will be told whether the hunk succeeded or failed, and which line (in the new file) *patch* thought the hunk should go on. If this is different from the line number specified in the diff you will be told the offset. A single large offset MAY be an indication that a hunk was installed in the wrong place. You will also be told if a fuzz factor was used to make the match, in which case you should also be slightly suspicious.

If no original file is specified on the command line, *patch* will try to figure out from the leading garbage what the name of the file to edit is. In the header of a context diff, the filename is found from lines beginning with "***" or "---", with the shortest name of an existing file winning. Only context diffs have lines like that, but if there is an "Index:" line in the leading garbage, *patch* will try to use the filename from that line. The context diff header takes precedence over an Index line. If no filename can be intuited from the leading garbage, you will be asked for the name of the file to patch.

(If the original file cannot be found, but a suitable SCCS or RCS file is handy, *patch* will attempt to get or check out the file.)

Additionally, if the leading garbage contains a "Prereq: " line, *patch* will take the first word from the prerequisites line (normally a version number) and check the input file to see if that word can be found. If not, *patch* will ask for confirmation before proceeding.

The upshot of all this is that you should be able to say, while in a news interface, the following:

```
| patch -d /usr/src/local/blurfl
```

and patch a file in the blurfl directory directly from the article containing the patch.

If the patch file contains more than one patch, *patch* will try to apply each of them as if they came from separate patch files. This means, among other things, that it is assumed that the name of the file to patch must be determined for each diff listing, and that the garbage before each diff listing will be examined for interesting things such as filenames and revision level, as mentioned previously. You can give switches (and another original file name) for the second and subsequent patches by separating the corresponding argument lists by a '+'. (The argument list for a second or subsequent patch may not specify a new patch file, however.)

Patch recognizes the following switches:

–b causes the next argument to be interpreted as the backup extension, to be used in place of ".orig".

–c forces *patch* to interpret the patch file as a context diff.

–d causes *patch* to interpret the next argument as a directory, and cd to it before doing anything else.

–D causes *patch* to use the "#ifdef...#endif" construct to mark changes. The argument following will be used as the differentiating symbol. Note that, unlike the C compiler, there must be a space between the **–D** and the argument.

–e forces *patch* to interpret the patch file as an ed script.

–f forces *patch* to assume that the user knows exactly what he or she is doing, and to not ask any questions. It does not suppress commentary, however. Use **–s** for that.

–F <number>
sets the maximum fuzz factor. This switch only applied to context diffs, and causes *patch* to ignore up to that many lines in looking for places to install a hunk. Note that a larger fuzz factor increases the odds of a faulty patch. The default fuzz factor is 2, and it may not be set to more than the number of lines of context in the context diff, ordinarily 3.

–l causes the pattern matching to be done loosely, in case the tabs and spaces have been munged in your input file. Any sequence of whitespace in the pattern line will match any sequence in the input file. Normal characters must still match exactly. Each line of the context must still match a line in the input file.

–n forces *patch* to interpret the patch file as a normal diff.

–N causes *patch* to ignore patches that it thinks are reversed or already applied. See also **–R .**

–o causes the next argument to be interpreted as the output file name.

–p <number>
sets the pathname strip count, which controls how pathnames found in the patch file are treated, in case the you keep your files in a different directory than the person who sent out the patch. The strip count specifies how many backslashes are to be stripped from the front of the pathname. (Any intervening directory names also go away.) For example, supposing the filename in the patch file was

/u/howard/src/blurfl/blurfl.c

setting **–p** or **–p0** gives the entire pathname unmodified, **–p1** gives

u/howard/src/blurfl/blurfl.c

without the leading slash, **–p4** gives

blurfl/blurfl.c

and not specifying **–p** at all just gives you "blurfl.c". Whatever you end up with is looked for either in the current directory, or the directory specified by the **–d** switch.

–r causes the next argument to be interpreted as the reject file name.

–R tells *patch* that this patch was created with the old and new files swapped. (Yes, I'm afraid that does happen occasionally, human nature being what it is.) *Patch* will attempt to swap each hunk around before applying it. Rejects will come out in the swapped format. The **–R** switch will not work with ed diff scripts because there is too little information to reconstruct the reverse operation.

If the first hunk of a patch fails, *patch* will reverse the hunk to see if it can be applied that way. If it can, you will be asked if you want to have the **–R** switch set. If it can't, the patch will continue to be applied normally. (Note: this method cannot detect a reversed patch if it is a normal diff and if the first command is an append (i.e. it should have been a delete) since appends always succeed, due to the fact that a null context will match anywhere. Luckily, most patches add or change lines rather than delete them, so most reversed normal diffs will begin with a delete, which will fail, triggering the heuristic.)

–s makes *patch* do its work silently, unless an error occurs.

–S causes *patch* to ignore this patch from the patch file, but continue on looking for the next patch in the file. Thus

patch -S + -S + <patchfile

will ignore the first and second of three patches.

–v causes *patch* to print out it's revision header and patch level.

–x<number>
sets internal debugging flags, and is of interest only to *patch* patchers.

ENVIRONMENT
No environment variables are used by *patch*.

FILES
/tmp/patch*

SEE ALSO
diff(1)

AUTHOR
Larry Wall

NOTES FOR PATCH SENDERS
There are several things you should bear in mind if you are going to be sending out patches. First, you can save people a lot of grief by keeping a patchlevel.h file which is patched to increment the patch level as the first diff in the patch file you send out. If you put a Prereq: line in with the patch, it won't let them apply patches out of order without some warning. Second, make sure you've specified the filenames right, either in a context diff header, or with an Index:

line. If you are patching something in a subdirectory, be sure to tell the patch user to specify a **-p** switch as needed. Third, you can create a file by sending out a diff that compares a null file to the file you want to create. This will only work if the file you want to create doesn't exist already in the target directory. Fourth, take care not to send out reversed patches, since it makes people wonder whether they already applied the patch. Fifth, while you may be able to get away with putting 582 diff listings into one file, it is probably wiser to group related patches into separate files in case something goes haywire.

DIAGNOSTICS

Generally indicative that *patch* couldn't parse your patch file.

The message "Hmm..." indicates that there is unprocessed text in the patch file and that *patch* is attempting to intuit whether there is a patch in that text and, if so, what kind of patch it is.

CAVEATS

Patch cannot tell if the line numbers are off in an ed script, and can only detect bad line numbers in a normal diff when it finds a "change" or a "delete" command. A context diff using fuzz factor 3 may have the same problem. Until a suitable interactive interface is added, you should probably do a context diff in these cases to see if the changes made sense. Of course, compiling without errors is a pretty good indication that the patch worked, but not always.

Patch usually produces the correct results, even when it has to do a lot of guessing. However, the results are guaranteed to be correct only when the patch is applied to exactly the same version of the file that the patch was generated from.

BUGS

Could be smarter about partial matches, excessively deviant offsets and swapped code, but that would take an extra pass.

If code has been duplicated (for instance with #ifdef OLDCODE ... #else ... #endif), *patch* is incapable of patching both versions, and, if it works at all, will likely patch the wrong one, and tell you that it succeeded to boot.

If you apply a patch you've already applied, *patch* will think it is a reversed patch, and offer to un-apply the patch. This could be construed as a feature.

NAME

pax – portable archive exchange

SYNOPSIS

pax [**–cv**] [**–f** *archive*] [**–t** *device*] [*pattern...*]

pax –r [**–cdimnopuvy**] [**–f** *archive*] [**–s** *replstr*] [**–t** *device*] [*pattern...*]

pax –w [**–aimuvy**] [**–b** *blocking*] [**–f** *archive*] [**–s** *replstr*] [**–t** *device*] [**–x** *format*] [*pathname...*]

pax –rw [**–ilmopuvy**] [**–s** *replstr*] [*pathname...*] directory

DESCRIPTION

Pax reads and writes archive files which conform to the **Archive/Interchange File Format** specified in *IEEE Std. 1003.1-1988*. *Pax* can also read, but not write, a number of other file formats in addition to those specified in the **Archive/Interchange File Format** description. Support for these traditional file formats, such as V7 *tar* and System V binary *cpio* format archives, is provided for backward compatibility and to maximize portability.

Pax will also support traditional *cpio* and System V *tar* interfaces if invoked with the name "cpio" or "tar" respectively. See the *cpio(1)* or *tar(1)* manual pages for more details.

Combinations of the **–r** and **–w** command line arguments specify whether *pax* will read, write or list the contents of the specified archive, or move the specified files to another directory.

The command line arguments are:

–w writes the files and directories specified by *pathname* operands to the standard output together with the pathname and status information prescribed by the archive format used. A directory *pathname* operand refers to the files and (recursively) subdirectories of that directory. If no *pathname* operands are given, then the standard input is read to get a list of pathnames to copy, one pathname per line. In this case, only those pathnames appearing on the standard input are copied. If *pathname* is given it must be the last operand.

–r *Pax* reads an archive file from the standard input. Only files with names that match any of the *pattern* operands are selected for extraction. The selected files are conditionally created and copied relative to the current directory tree, subject to the options described below. By default, the owner and group of selected files will be that of the invoking process, and the permissions and modification times will be the sames as those in the archive.

The supported archive formats are automatically detected on input. The default output format is *ustar*, but may be overridden by the **–x** *format* option described below.

–rw *Pax* reads the files and directories named in the *pathname* operands and copies them to the destination *directory*. A directory *pathname* operand refers to the files and (recursively) subdirectories of that directory. If no *pathname* operands are given, the standard input is read to get a list of pathnames to copy, one pathname per line. In this case, only those pathnames appearing on the standard input are copied. The directory named by the *directory* operand must exist and have the proper permissions before the copy can occur.

If neither the **–r** or **–w** options are given, then *pax* will list the contents of the specified archive. In this mode, *pax* lists normal files one per line, hard link pathnames as

pathname == *linkname*

and symbolic link pathnames (if supported by the implementation) as

pathname -> *linkname*

where *pathname* is the name of the file being extracted, and *linkname* is the name of a file which appeared earlier in the archive.

If the **–v** option is specified, then *pax* list normal pathnames in the same format used by the *ls* utility with the **–l** option. Hard links are shown as

<*ls -l listing*> == *linkname*

and symbolic links (if supported) are shown as

<ls -l listing> -> linkname

Options
The following options are available:

–a The files specified by *pathname* are appended to the specified archive.

–b *blocking* Block the output at *blocking* bytes per write to the archive file. A **k** suffix multiplies *blocking* by 1024, a **b** suffix multiplies *blocking* by 512 and a **m** suffix multiplies *blocking* by 1048576 (1 megabyte). If not specified, *blocking* is automatically determined on input and is ignored for **–rw.**

–c Complement the match sense of the *pattern* operands.

–d Intermediate directories not explicitly listed in the archive are not created. This option is ignored unless the **–r** option is specified.

–f *archive* The *archive* option specifies the pathname of the input or output archive, overriding the default of standard input for **–r** or standard output for **–w**.

–i Interactively rename files. Substitutions specified by **–s** options (described below) are performed before requesting the new file name from the user. A file is skipped if an empty line is entered and *pax* exits with an exit status of 0 if **EOF** is encountered.

–l Files are linked rather than copied when possible.

–m File modification times are not retained.

–n When **–r** is specified, but **–w** is not, the *pattern* arguments are treated as ordinary file names. Only the first occurrence of each of these files in the input archive is read. The **pax** utility exits with a zero exit status after all files in the list have been read. If one or more files in the list is not found, **pax** writes a diagnostic to standard error for each of the files and exits with a non-zero exit status. the file names are compared before any of the **–i**, **–s**, or **–y** options are applied.

–o Restore file ownership as specified in the archive. Thc invoking process must have appropriate privileges to accomplish this.

–p Preserve the access time of the input files after they have been copied.

–s *replstr* File names are modified according to the substitution expression using the syntax of *ed(1)* as shown:

-s /*old*/*new*/[gp]

Any non null character may be used as a delimiter (a / is used here as an example). Multiple **–s** expressions may be specified; the expressions are applied in the order specified terminating with the first successful substitution. The optional trailing **p** causes successful mappings to be listed on standard error. The optional trailing **g** causes the *old* expression to be replaced each time it occurs in the source string. Files that substitute to an empty string are ignored both on input and output.

–t *device* The *device* option argument is an implementation-defined identifier that names the input or output archive device, overriding the default of standard input for **–r** and standard output for **–w**.

–u Copy each file only if it is newer than a pre-existing file with the same name. This implies **–a**.

–v List file names as they are encountered. Produces a verbose table of contents listing on the standard output when both **–r** and **–w** are omitted, otherwise the file names are printed to standard error as they are encountered in the archive.

–x *format* Specifies the output archive *format*. The input format, which must be one of the

following, is automatically determined when the **–r** option is used. The supported formats are:

cpio The extended *CPIO* interchange format specified in **Extended CPIO Format** in *IEEE Std. 1003.1-1988.*

ustar The extended *TAR* interchange format specified in **Extended TAR Format in** *IEEE Std. 1003.1-1988.* This is the default archive format.

–y Interactively prompt for the disposition of each file. Substitutions specified by **–s** options (described above) are performed before prompting the user for disposition. **EOF** or an input line starting with the character **q** caused *pax* to exit. Otherwise, an input line starting with anything other than **y** causes the file to be ignored. This option cannot be used in conjunction with the **–i** option.

Only the last of multiple **–f** or **–t** options take effect.

When writing to an archive, the standard input is used as a list of pathnames if no *pathname* operands are specified. The format is one pathname per line. Otherwise, the standard input is the archive file, which is formatted according to one of the specifications in **Archive/Interchange File format** in *IEEE Std. 1003.1-1988*, or some other implementation-defined format.

The user ID and group ID of the process, together with the appropriate privileges, affect the ability of *pax* to restore ownership and permissions attributes of the archived files. (See *format-reading utility* in **Archive/Interchange File Format** in *IEEE Std. 1003.1-1988.*)

The options **–a**, **–c**, **–d**, **–i**, **–l**, **–p**, **–t**, **–u**, and **–y** are provided for functional compatibility with the historical *cpio* and *tar* utilities. The option defaults were chosen based on the most common usage of these options, therefore, some of the options have meanings different than those of the historical commands.

Operands

The following operands are available:

directory The destination directory pathname for copies when both the **–r** and **–w** options are specified. The directory must exist and be writable before the copy or and error results.

pathname A file whose contents are used instead of the files named on the standard input. When a directory is named, all of its files and (recursively) subdirectories are copied as well.

pattern A *pattern* is given in the standard shell pattern matching notation. The default if no *pattern* is specified is *, which selects all files.

EXAMPLES

The following command

```
pax –w –f /dev/rmt0 .
```

copies the contents of the current directory to tape drive 0.

The commands

```
mkdir newdir
cd olddir
pax -rw . newdir
```

copies the contents of *olddir* to *newdir* .

The command

```
pax –r –s ',//*usr//*,,' -f pax.out
```

reads the archive **pax.out** with all files rooted in "/usr" in the archive extracted relative to the current directory.

FILES

/dev/tty used to prompt the user for information when the **–i** or **–y** options are specified.

pax is capable of archiving files and generating archives that are greater than two gigabytes in size. *pax* limits the size of the files to be archived to eight gigabytes, ignoring larger files with a warning.

SEE ALSO

cpio(1), find(1), tar(1), cpio(5), tar(5)

DIAGNOSTICS

Pax will terminate immediately, without processing any additional files on the command line or in the archive.

EXIT CODES

Pax will exit with one of the following values:

0 All files in the archive were processed successfully.

>0 *Pax* aborted due to errors encountered during operation.

BUGS

Special permissions may be required to copy or extract special files.

Device, user ID, and group ID numbers larger than 65535 cause additional header records to be output. These records are ignored by some historical version of *cpio(1)* and *tar(1)*.

The archive formats described in **Archive/Interchange File Format** have certain restrictions that have been carried over from historical usage. For example, there are restrictions on the length of pathnames stored in the archive.

When getting an "ls -l" style listing on *tar* format archives, link counts are listed as zero since the *ustar* archive format does not keep link count information.

COPYRIGHT

AUTHOR

Mark H. Colburn
NAPS International
117 Mackubin Street, Suite 1
St. Paul, MN 55102
mark@jhereg.MN.ORG

Sponsored by **The USENIX Association** for public distribution.

NAME

perl – Practical Extraction and Report Language

SYNOPSIS

perl [options] filename args

DESCRIPTION

Perl is an interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It's also a good language for many system management tasks. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). It combines (in the author's opinion, anyway) some of the best features of C, *sed*, *awk*, and *sh*, so people familiar with those languages should have little difficulty with it. (Language historians will also note some vestiges of *csh*, Pascal, and even BASIC-PLUS.) Expression syntax corresponds quite closely to C expression syntax. Unlike most Unix utilities, *perl* does not arbitrarily limit the size of your data—if you've got the memory, *perl* can slurp in your whole file as a single string. Recursion is of unlimited depth. And the hash tables used by associative arrays grow as necessary to prevent degraded performance. *Perl* uses sophisticated pattern matching techniques to scan large amounts of data very quickly. Although optimized for scanning text, *perl* can also deal with binary data, and can make dbm files look like associative arrays (where dbm is available). Setuid *perl* scripts are safer than C programs through a dataflow tracing mechanism which prevents many stupid security holes. If you have a problem that would ordinarily use *sed* or *awk* or *sh*, but it exceeds their capabilities or must run a little faster, and you don't want to write the silly thing in C, then *perl* may be for you. There are also translators to turn your *sed* and *awk* scripts into *perl* scripts. OK, enough hype.

Upon startup, *perl* looks for your script in one of the following places:

1. Specified line by line via **–e** switches on the command line.
2. Contained in the file specified by the first filename on the command line. (Note that systems supporting the #! notation invoke interpreters this way.)
3. Passed in implicitly via standard input. This only works if there are no filename arguments—to pass arguments to a *stdin* script you must explicitly specify a – for the script name.

After locating your script, *perl* compiles it to an internal form. If the script is syntactically correct, it is executed.

Options

Note: on first reading this section may not make much sense to you. It's here at the front for easy reference.

A single-character option may be combined with the following option, if any. This is particularly useful when invoking a script using the #! construct which only allows one argument. Example:

```
#!/usr/bin/perl –spi.bak        # same as –s –p –i.bak
...
```

Options include:

–0*digits*

specifies the record separator ($/) as an octal number. If there are no digits, the null character is the separator. Other switches may precede or follow the digits. For example, if you have a version of *find* which can print filenames terminated by the null character, you can say this:

```
find . –name '*.bak' –print0 | perl –n0e unlink
```

The special value 00 will cause Perl to slurp files in paragraph mode. The value 0777 will cause Perl to slurp files whole since there is no legal character with that value.

-a turns on autosplit mode when used with a **-n** or **-p**. An implicit split command to the @F array is done as the first thing inside the implicit while loop produced by the **-n** or **-p**.

```
perl -ane 'print pop(@F), "\n";'
```

is equivalent to

```
while (<>) {
        @F = split(' ');
        print pop(@F), "\n";
}
```

-c causes *perl* to check the syntax of the script and then exit without executing it.

-d runs the script under the perl debugger. See the section on Debugging.

-D*number*

sets debugging flags. To watch how it executes your script, use **-D14**. (This only works if debugging is compiled into your *perl*.) Another nice value is -D1024, which lists your compiled syntax tree. And -D512 displays compiled regular expressions.

-e *commandline*

may be used to enter one line of script. Multiple **-e** commands may be given to build up a multi-line script. If **-e** is given, *perl* will not look for a script filename in the argument list.

-i*extension*

specifies that files processed by the <> construct are to be edited in-place. It does this by renaming the input file, opening the output file by the same name, and selecting that output file as the default for print statements. The extension, if supplied, is added to the name of the old file to make a backup copy. If no extension is supplied, no backup is made. Saying "perl -p -i.bak -e "s/foo/bar/;" ... " is the same as using the script:

```
#!/usr/bin/perl -pi.bak
s/foo/bar/;
```

which is equivalent to

```
#!/usr/bin/perl
while (<>) {
        if ($ARGV ne $oldargv) {
                rename($ARGV, $ARGV . '.bak');
                open(ARGVOUT, ">$ARGV");
                select(ARGVOUT);
                $oldargv = $ARGV;
        }
        s/foo/bar/;
}
continue {
    print;          # this prints to original filename
}
select(STDOUT);
```

except that the **-i** form doesn't need to compare $ARGV to $oldargv to know when the

filename has changed. It does, however, use ARGVOUT for the selected filehandle. Note that *STDOUT* is restored as the default output filehandle after the loop.

You can use eof to locate the end of each input file, in case you want to append to each file, or reset line numbering (see example under eof).

-I*directory*
may be used in conjunction with **-P** to tell the C preprocessor where to look for include files. By default /usr/include and /usr/lib/perl are searched.

-l*octnum*
enables automatic line-ending processing. It has two effects: first, it automatically chops the line terminator when used with **-n** or **-p** , and second, it assigns $\ to have the value of *octnum* so that any print statements will have that line terminator added back on. If *octnum* is omitted, sets $\ to the current value of $/. For instance, to trim lines to 80 columns:

```
perl -lpe 'substr($_, 80) = ""'
```

Note that the assignment $\ = $/ is done when the switch is processed, so the input record separator can be different than the output record separator if the **-l** switch is followed by a **-0** switch:

```
gnufind / -print0 | perl -ln0e 'print "found $_" if -p'
```

This sets $\ to newline and then sets $/ to the null character.

-n causes *perl* to assume the following loop around your script, which makes it iterate over filename arguments somewhat like "sed -n" or *awk*:

```
while (<>) {
        ...             # your script goes here
}
```

Note that the lines are not printed by default. See **-p** to have lines printed. Here is an efficient way to delete all files older than a week:

```
find . -mtime +7 -print | perl -nle 'unlink;'
```

This is faster than using the -exec switch of find because you don't have to start a process on every filename found.

-p causes *perl* to assume the following loop around your script, which makes it iterate over filename arguments somewhat like *sed*:

```
while (<>) {
        ...             # your script goes here
} continue {
        print;
}
```

Note that the lines are printed automatically. To suppress printing use the **-n** switch. A **-p** overrides a **-n** switch.

-P causes your script to be run through the C preprocessor before compilation by *perl*. (Since both comments and cpp directives begin with the # character, you should avoid starting comments with any words recognized by the C preprocessor such as "if", "else" or

"define".)

–s enables some rudimentary switch parsing for switches on the command line after the script name but before any filename arguments (or before a --). Any switch found there is removed from @ARGV and sets the corresponding variable in the *perl* script. The following script prints "true" if and only if the script is invoked with a –xyz switch.

```
#!/usr/bin/perl –s
if ($xyz) { print "true\n"; }
```

–S makes *perl* use the PATH environment variable to search for the script (unless the name of the script starts with a slash). Typically this is used to emulate #! startup on machines that don't support #!, in the following manner:

```
#!/usr/bin/perl
eval "exec /usr/bin/perl –S $0 $*"
        if $running_under_some_shell;
```

The system ignores the first line and feeds the script to /bin/sh, which proceeds to try to execute the *perl* script as a shell script. The shell executes the second line as a normal shell command, and thus starts up the *perl* interpreter. On some systems $0 doesn't always contain the full pathname, so the **–S** tells *perl* to search for the script if necessary. After *perl* locates the script, it parses the lines and ignores them because the variable $running_under_some_shell is never true. A better construct than $* would be ${1+"$@"}, which handles embedded spaces and such in the filenames, but doesn't work if the script is being interpreted by csh. In order to start up sh rather than csh, some systems may have to replace the #! line with a line containing just a colon, which will be politely ignored by perl. Other systems can't control that, and need a totally devious construct that will work under any of csh, sh or perl, such as the following:

```
eval '(exit $?0)' && eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'
& eval 'exec /usr/bin/perl -S $0 $argv:q'
        if 0;
```

–u causes *perl* to dump core after compiling your script. You can then take this core dump and turn it into an executable file by using the undump program (not supplied). This speeds startup at the expense of some disk space (which you can minimize by stripping the executable). (Still, a "hello world" executable comes out to about 200K on my machine.) If you are going to run your executable as a set-id program then you should probably compile it using taintperl rather than normal perl. If you want to execute a portion of your script before dumping, use the dump operator instead. Note: availability of undump is platform specific and may not be available for a specific port of perl. On a CONVEX system, instead of an actual coredump, the GNUemacs unexec() function is called to create an executable file called *script*.perldump (where *script* is the name of your script), which can then be run directly.

–U allows *perl* to do unsafe operations. Currently the only "unsafe" operation is the unlinking of directories while running as superuser.

–v prints the version and patchlevel of your *perl* executable.

–w prints warnings about identifiers that are mentioned only once, and scalar variables that are used before being set. Also warns about redefined subroutines, and references to undefined filehandles or filehandles opened readonly that you are attempting to write on. Also warns you if you use == on values that don't look like numbers, and if your subroutines recurse

more than 100 deep.

-xdirectory

tells *perl* that the script is embedded in a message. Leading garbage will be discarded until the first line that starts with #! and contains the string "perl". Any meaningful switches on that line will be applied (but only one group of switches, as with normal #! processing). If a directory name is specified, Perl will switch to that directory before running the script. The -x switch only controls the the disposal of leading garbage. The script must be terminated with __END__ if there is trailing garbage to be ignored (the script can process any or all of the trailing garbage via the DATA filehandle if desired).

Data Types and Objects

Perl has three data types: scalars, arrays of scalars, and associative arrays of scalars. Normal arrays are indexed by number, and associative arrays by string.

The interpretation of operations and values in perl sometimes depends on the requirements of the context around the operation or value. There are three major contexts: string, numeric and array. Certain operations return array values in contexts wanting an array, and scalar values otherwise. (If this is true of an operation it will be mentioned in the documentation for that operation.) Operations which return scalars don't care whether the context is looking for a string or a number, but scalar variables and values are interpreted as strings or numbers as appropriate to the context. A scalar is interpreted as TRUE in the boolean sense if it is not the null string or 0. Booleans returned by operators are 1 for true and 0 or '' (the null string) for false.

There are actually two varieties of null string: defined and undefined. Undefined null strings are returned when there is no real value for something, such as when there was an error, or at end of file, or when you refer to an uninitialized variable or element of an array. An undefined null string may become defined the first time you access it, but prior to that you can use the defined() operator to determine whether the value is defined or not.

References to scalar variables always begin with '$', even when referring to a scalar that is part of an array. Thus:

```
$days               # a simple scalar variable
$days[28]           # 29th element of array @days
$days{'Feb'}        # one value from an associative array
$#days              # last index of array @days
```

but entire arrays or array slices are denoted by '@':

```
@days               # ($days[0], $days[1],... $days[n])
@days[3,4,5]        # same as @days[3..5]
@days{'a','c'}      # same as ($days{'a'},$days{'c'})
```

and entire associative arrays are denoted by '%':

```
%days               # (key1, val1, key2, val2 ...)
```

Any of these eight constructs may serve as an lvalue, that is, may be assigned to. (It also turns out that an assignment is itself an lvalue in certain contexts—see examples under s, tr and chop.) Assignment to a scalar evaluates the righthand side in a scalar context, while assignment to an array or array slice evaluates the righthand side in an array context.

You may find the length of array @days by evaluating "$#days", as in *csh*. (Actually, it's not the length of the array, it's the subscript of the last element, since there is (ordinarily) a 0th element.) Assigning to $#days changes the length of the array. Shortening an array by this method does not actually destroy any values. Lengthening an array that was previously

shortened recovers the values that were in those elements. You can also gain some measure of efficiency by preextending an array that is going to get big. (You can also extend an array by assigning to an element that is off the end of the array. This differs from assigning to $#whatever in that intervening values are set to null rather than recovered.) You can truncate an array down to nothing by assigning the null list () to it. The following are exactly equivalent

```
@whatever = ();
$#whatever = $[ - 1;
```

If you evaluate an array in a scalar context, it returns the length of the array. The following is always true:

```
@whatever === $#whatever - $[ + 1;
```

Multi-dimensional arrays are not directly supported, but see the discussion of the $; variable later for a means of emulating multiple subscripts with an associative array. You could also write a subroutine to turn multiple subscripts into a single subscript.

Every data type has its own namespace. You can, without fear of conflict, use the same name for a scalar variable, an array, an associative array, a filehandle, a subroutine name, and/or a label. Since variable and array references always start with '$', '@', or '%', the "reserved" words aren't in fact reserved with respect to variable names. (They ARE reserved with respect to labels and filehandles, however, which don't have an initial special character. Hint: you could say open(LOG,'logfile') rather than open(log,'logfile'). Using uppercase filehandles also improves readability and protects you from conflict with future reserved words.) Case IS significant—"FOO", "Foo" and "foo" are all different names. Names which start with a letter may also contain digits and underscores. Names which do not start with a letter are limited to one character, e.g. "$%" or "$$". (Most of the one character names have a predefined significance to *perl*. More later.)

Numeric literals are specified in any of the usual floating point or integer formats:

```
12345
12345.67
.23E-10
0xffff # hex
0377 # octal
```

String literals are delimited by either single or double quotes. They work much like shell quotes: double-quoted string literals are subject to backslash and variable substitution; single-quoted strings are not (except for \' and \\). The usual backslash rules apply for making characters such as newline, tab, etc., as well as some more exotic forms:

\t	tab
\n	newline
\r	return
\f	form feed
\b	backspace
\a	alarm (bell)
\e	escape
\033	octal char
\x1b	hex char
\c[	control char
\l	lowercase next char

\u	uppercase next char
\L	lowercase till \E
\U	uppercase till \E
\E	end case modification

You can also embed newlines directly in your strings, i.e. they can end on a different line than they begin. This is nice, but if you forget your trailing quote, the error will not be reported until *perl* finds another line containing the quote character, which may be much further on in the script. Variable substitution inside strings is limited to scalar variables, normal array values, and array slices. (In other words, identifiers beginning with $ or @, followed by an optional bracketed expression as a subscript.) The following code segment prints out "The price is $100."

```
$Price = '$100';                    # not interpreted
print "The price is $Price.\n";      # interpreted
```

Note that you can put curly brackets around the identifier to delimit it from following alphanumerics. Also note that a single quoted string must be separated from a preceding word by a space, since single quote is a valid character in an identifier (see Packages).

Two special literals are __LINE__ and __FILE__, which represent the current line number and filename at that point in your program. They may only be used as separate tokens; they will not be interpolated into strings. In addition, the token __END__ may be used to indicate the logical end of the script before the actual end of file. Any following text is ignored (but may be read via the DATA filehandle). The two control characters ^D and ^Z are synonyms for __END__.

A word that doesn't have any other interpretation in the grammar will be treated as if it had single quotes around it. For this purpose, a word consists only of alphanumeric characters and underline, and must start with an alphabetic character. As with filehandles and labels, a bare word that consists entirely of lowercase letters risks conflict with future reserved words, and if you use the –w switch, Perl will warn you about any such words.

Array values are interpolated into double-quoted strings by joining all the elements of the array with the delimiter specified in the $" variable, space by default. (Since in versions of perl prior to 3.0 the @ character was not a metacharacter in double-quoted strings, the interpolation of @array, $array[EXPR], @array[LIST], $array{EXPR}, or @array{LIST} only happens if array is referenced elsewhere in the program or is predefined.) The following are equivalent:

```
$temp = join($",@ARGV);
system "echo $temp";

system "echo @ARGV";
```

Within search patterns (which also undergo double-quotish substitution) there is a bad ambiguity: Is /$foo[bar]/ to be interpreted as /${foo}[bar]/ (where [bar] is a character class for the regular expression) or as /${foo[bar]}/ (where [bar] is the subscript to array @foo)? If @foo doesn't otherwise exist, then it's obviously a character class. If @foo exists, perl takes a good guess about [bar], and is almost always right. If it does guess wrong, or if you're just plain paranoid, you can force the correct interpretation with curly brackets as above.

A line-oriented form of quoting is based on the shell here-is syntax. Following a << you specify a string to terminate the quoted material, and all lines following the current line down to the terminating string are the value of the item. The terminating string may be either an identifier (a word), or some quoted text. If quoted, the type of quotes you use determines the treatment of the text, just as in regular quoting. An unquoted identifier works like double quotes. There must be no space between the << and the identifier. (If you put a space it will be treated as a null identifier, which is valid, and matches the first blank line—see Merry Christmas example below.)

The terminating string must appear by itself (unquoted and with no surrounding whitespace) on the terminating line.

```
        print <<EOF;          # same as above
The price is $Price.
EOF

        print <<"EOF";                # same as above
The price is $Price.
EOF

        print << x 10;        # null identifier is delimiter
Merry Christmas!

        print <<`EOC`;                # execute commands
echo hi there
echo lo there
EOC

        print <<foo, <<bar; # you can stack them
I said foo.
foo
I said bar.
bar
```

Array literals are denoted by separating individual values by commas, and enclosing the list in parentheses:

```
        (LIST)
```

In a context not requiring an array value, the value of the array literal is the value of the final element, as in the C comma operator. For example,

```
@foo = ('cc', '-E', $bar);
```

assigns the entire array value to array foo, but

```
$foo = ('cc', '-E', $bar);
```

assigns the value of variable bar to variable foo. Note that the value of an actual array in a scalar context is the length of the array; the following assigns to $foo the value 3:

```
@foo = ('cc', '-E', $bar);
$foo = @foo;                  # $foo gets 3
```

You may have an optional comma before the closing parenthesis of an array literal, so that you can say:

```
@foo = (
    1,
    2,
    3,
);
```

When a LIST is evaluated, each element of the list is evaluated in an array context, and the resulting array value is interpolated into LIST just as if each individual element were a member of LIST. Thus arrays lose their identity in a LIST—the list

```
(@foo,@bar,&SomeSub)
```

contains all the elements of @foo followed by all the elements of @bar, followed by all the elements returned by the subroutine named SomeSub.

A list value may also be subscripted like a normal array. Examples:

```
$time = (stat($file))[8];  # stat returns array value
$digit = ('a','b','c','d','e','f')[$digit-10];
return (pop(@foo),pop(@foo))[0];
```

Array lists may be assigned to if and only if each element of the list is an lvalue:

```
($a, $b, $c) = (1, 2, 3);

($map{'red'}, $map{'blue'}, $map{'green'}) = (0x00f, 0x0f0, 0xf00);
```

The final element may be an array or an associative array:

```
($a, $b, @rest) = split;
local($a, $b, %rest) = @_;
```

You can actually put an array anywhere in the list, but the first array in the list will soak up all the values, and anything after it will get a null value. This may be useful in a local().

An associative array literal contains pairs of values to be interpreted as a key and a value:

```
# same as map assignment above
%map = ('red',0x00f,'blue',0x0f0,'green',0xf00);
```

Array assignment in a scalar context returns the number of elements produced by the expression on the right side of the assignment:

```
$x = (($foo,$bar) = (3,2,1));     # set $x to 3, not 2
```

There are several other pseudo-literals that you should know about. If a string is enclosed by backticks (grave accents), it first undergoes variable substitution just like a double quoted string. It is then interpreted as a command, and the output of that command is the value of the pseudo-literal, like in a shell. In a scalar context, a single string consisting of all the output is returned. In an array context, an array of values is returned, one for each line of output. (You can set $/ to use a different line terminator.) The command is executed each time the pseudo-literal is evaluated. The status value of the command is returned in $? (see Predefined Names for the interpretation of $?). Unlike in *csh*, no translation is done on the return data—newlines remain newlines. Unlike in any of the shells, single quotes do not hide variable names in the command from interpretation. To pass a $ through to the shell you need to hide it with a backslash.

Evaluating a filehandle in angle brackets yields the next line from that file (newline included, so it's never false until EOF, at which time an undefined value is returned). Ordinarily you must assign that value to a variable, but there is one situation where an automatic assignment happens. If (and only if) the input symbol is the only thing inside the conditional of a *while* loop, the

value is automatically assigned to the variable "$_". (This may seem like an odd thing to you, but you'll use the construct in almost every *perl* script you write.) Anyway, the following lines are equivalent to each other:

```
while ($_ = <STDIN>) { print; }
while (<STDIN>) { print; }
for (;<STDIN>;) { print; }
print while $_ = <STDIN>;
print while <STDIN>;
```

The filehandles *STDIN*, *STDOUT* and *STDERR* are predefined. (The filehandles ***stdin***, ***stdout*** and ***stderr*** will also work except in packages, where they would be interpreted as local identifiers rather than global.) Additional filehandles may be created with the ***open*** function.

If a <FILEHANDLE> is used in a context that is looking for an array, an array consisting of all the input lines is returned, one line per array element. It's easy to make a LARGE data space this way, so use with care.

The null filehandle <> is special and can be used to emulate the behavior of *sed* and *awk*. Input from <> comes either from standard input, or from each file listed on the command line. Here's how it works: the first time <> is evaluated, the ARGV array is checked, and if it is null, $ARGV[0] is set to '-', which when opened gives you standard input. The ARGV array is then processed as a list of filenames. The loop

```
while (<>) {
        ...                     # code for each line
}
```

is equivalent to

```
unshift(@ARGV, '-') if $#ARGV < $[;
while ($ARGV = shift) {
        open(ARGV, $ARGV);
        while (<ARGV>) {
                ...             # code for each line
        }
}
```

except that it isn't as cumbersome to say. It really does shift array ARGV and put the current filename into variable ARGV. It also uses filehandle ARGV internally. You can modify @ARGV before the first <> as long as you leave the first filename at the beginning of the array. Line numbers ($.) continue as if the input was one big happy file. (But see example under eof for how to reset line numbers on each file.)

If you want to set @ARGV to your own list of files, go right ahead. If you want to pass switches into your script, you can put a loop on the front like this:

```
while ($_ = $ARGV[0], /^-/) {
        shift;
    last if /^--$/;
        /^-D(.*)/ && ($debug = $1);
        /^-v/ && $verbose++;
        ...             # other switches
}
while (<>) {
        ...             # code for each line
}
```

The <> symbol will return FALSE only once. If you call it again after this it will assume you are processing another @ARGV list, and if you haven't set @ARGV, will input from *STDIN*.

If the string inside the angle brackets is a reference to a scalar variable (e.g. <$foo>), then that variable contains the name of the filehandle to input from.

If the string inside angle brackets is not a filehandle, it is interpreted as a filename pattern to be globbed, and either an array of filenames or the next filename in the list is returned, depending on context. One level of $ interpretation is done first, but you can't say <$foo> because that's an indirect filehandle as explained in the previous paragraph. You could insert curly brackets to force interpretation as a filename glob: <${foo}>. Example:

```
while (<*.c>) {
        chmod 0644, $_;
}
```

is equivalent to

```
open(foo, "echo *.c | tr -s ' \t\r\f' '\\012\\012\\012\\012'|");
while (<foo>) {
        chop;
        chmod 0644, $_;
}
```

In fact, it's currently implemented that way. (Which means it will not work on filenames with spaces in them unless you have /bin/csh on your machine.) Of course, the shortest way to do the above is:

```
chmod 0644, <*.c>;
```

Syntax

A *perl* script consists of a sequence of declarations and commands. The only things that need to be declared in *perl* are report formats and subroutines. See the sections below for more information on those declarations. All uninitialized user-created objects are assumed to start with a null or 0 value until they are defined by some explicit operation such as assignment. The sequence of commands is executed just once, unlike in *sed* and *awk* scripts, where the sequence of commands is executed for each input line. While this means that you must explicitly loop over the lines of your input file (or files), it also means you have much more control over which files and which lines you look at. (Actually, I'm lying—it is possible to do an implicit loop with either the **-n** or **-p** switch.)

A declaration can be put anywhere a command can, but has no effect on the execution of the primary sequence of commands—declarations all take effect at compile time. Typically all the

declarations are put at the beginning or the end of the script.

Perl is, for the most part, a free-form language. (The only exception to this is format declarations, for fairly obvious reasons.) Comments are indicated by the # character, and extend to the end of the line. If you attempt to use /* */ C comments, it will be interpreted either as division or pattern matching, depending on the context. So don't do that.

Compound statements

In *perl*, a sequence of commands may be treated as one command by enclosing it in curly brackets. We will call this a BLOCK.

The following compound commands may be used to control flow:

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
LABEL while (EXPR) BLOCK
LABEL while (EXPR) BLOCK continue BLOCK
LABEL for (EXPR; EXPR; EXPR) BLOCK
LABEL foreach VAR (ARRAY) BLOCK
LABEL BLOCK continue BLOCK
```

Note that, unlike C and Pascal, these are defined in terms of BLOCKs, not statements. This means that the curly brackets are *required*—no dangling statements allowed. If you want to write conditionals without curly brackets there are several other ways to do it. The following all do the same thing:

```
if (!open(foo)) { die "Can't open $foo: $!"; }
die "Can't open $foo: $!" unless open(foo);
open(foo) || die "Can't open $foo: $!";   # foo or bust!
open(foo) ? 'hi mom' : die "Can't open $foo: $!";
                        # a bit exotic, that last one
```

The *if* statement is straightforward. Since BLOCKs are always bounded by curly brackets, there is never any ambiguity about which *if* an *else* goes with. If you use *unless* in place of *if*, the sense of the test is reversed.

The *while* statement executes the block as long as the expression is true (does not evaluate to the null string or 0). The LABEL is optional, and if present, consists of an identifier followed by a colon. The LABEL identifies the loop for the loop control statements *next*, *last*, and *redo* (see below). If there is a *continue* BLOCK, it is always executed just before the conditional is about to be evaluated again, similarly to the third part of a *for* loop in C. Thus it can be used to increment a loop variable, even when the loop has been continued via the *next* statement (similar to the C "continue" statement).

If the word *while* is replaced by the word *until*, the sense of the test is reversed, but the conditional is still tested before the first iteration.

In either the *if* or the *while* statement, you may replace "(EXPR)" with a BLOCK, and the conditional is true if the value of the last command in that block is true.

The *for* loop works exactly like the corresponding *while* loop:

```
for ($i = 1; $i < 10; $i++) {
        ...
}
```

is the same as

```
$i = 1;
while ($i < 10) {
        ...
} continue {
        $i++;
}
```

The foreach loop iterates over a normal array value and sets the variable VAR to be each element of the array in turn. The variable is implicitly local to the loop, and regains its former value upon exiting the loop. The "foreach" keyword is actually identical to the "for" keyword, so you can use "foreach" for readability or "for" for brevity. If VAR is omitted, $_ is set to each value. If ARRAY is an actual array (as opposed to an expression returning an array value), you can modify each element of the array by modifying VAR inside the loop. Examples:

```
for (@ary) { s/foo/bar/; }

foreach $elem (@elements) {
        $elem *= 2;
}

for ((10,9,8,7,6,5,4,3,2,1,'BOOM')) {
        print $_, "\n"; sleep(1);
}

for (1..15) { print "Merry Christmas\n"; }

foreach $item (split(/:[\\\n:]*/, $ENV{'TERMCAP'})) {
        print "Item: $item\n";
}
```

The BLOCK by itself (labeled or not) is equivalent to a loop that executes once. Thus you can use any of the loop control statements in it to leave or restart the block. The *continue* block is optional. This construct is particularly nice for doing case structures.

```
foo: {
        if (/^abc/) { $abc = 1; last foo; }
        if (/^def/) { $def = 1; last foo; }
        if (/^xyz/) { $xyz = 1; last foo; }
        $nothing = 1;
}
```

There is no official switch statement in perl, because there are already several ways to write the equivalent. In addition to the above, you could write

```
foo: {
        $abc = 1, last foo  if /^abc/;
        $def = 1, last foo  if /^def/;
        $xyz = 1, last foo  if /^xyz/;
        $nothing = 1;
}
```

or

```
foo: {
        /^abc/ && do { $abc = 1; last foo; };
        /^def/ && do { $def = 1; last foo; };
        /^xyz/ && do { $xyz = 1; last foo; };
        $nothing = 1;
}
```

or

```
foo: {
        /^abc/ && ($abc = 1, last foo);
        /^def/ && ($def = 1, last foo);
        /^xyz/ && ($xyz = 1, last foo);
        $nothing = 1;
}
```

or even

```
if (/^abc/)
        { $abc = 1; }
elsif (/^def/)
        { $def = 1; }
elsif (/^xyz/)
        { $xyz = 1; }
else
        {$nothing = 1;}
```

As it happens, these are all optimized internally to a switch structure, so perl jumps directly to the desired statement, and you needn't worry about perl executing a lot of unnecessary statements when you have a string of 50 elsifs, as long as you are testing the same simple scalar variable using ==, eq, or pattern matching as above. (If you're curious as to whether the optimizer has done this for a particular case statement, you can use the -D1024 switch to list the syntax tree before execution.)

Simple statements

The only kind of simple statement is an expression evaluated for its side effects. Every expression (simple statement) must be terminated with a semicolon. Note that this is like C, but unlike Pascal (and *awk*).

Any simple statement may optionally be followed by a single modifier, just before the terminating semicolon. The possible modifiers are:

```
if EXPR
unless EXPR
while EXPR
until EXPR
```

The *if* and *unless* modifiers have the expected semantics. The *while* and *until* modifiers also have the expected semantics (conditional evaluated first), except when applied to a do-BLOCK or a do-SUBROUTINE command, in which case the block executes once before the conditional is evaluated. This is so that you can write loops like:

```
do {
        $_ = <STDIN>;
        ...
} until $_ eq ".\n";
```

(See the *do* operator below. Note also that the loop control commands described later will NOT work in this construct, since modifiers don't take loop labels. Sorry.)

Expressions

Since *perl* expressions work almost exactly like C expressions, only the differences will be mentioned here.

Here's what *perl* has that C doesn't:

**

The exponentiation operator.

**=

The exponentiation assignment operator.

()

The null list, used to initialize an array to null.

.

Concatenation of two strings.

.=

The concatenation assignment operator.

eq

String equality (== is numeric equality). For a mnemonic just think of "eq" as a string. (If you are used to the *awk* behavior of using == for either string or numeric equality based on the current form of the comparands, beware! You must be explicit here.)

ne

String inequality (!= is numeric inequality).

lt

String less than.

gt

String greater than.

le

String less than or equal.

ge

String greater than or equal.

cmp

String comparison, returning -1, 0, or 1.

<=>

Numeric comparison, returning -1, 0, or 1.

=~

Certain operations search or modify the string "$_" by default. This operator makes that kind of operation work on some other string. The right argument is a search pattern, substitution, or translation. The left argument is what is supposed to be searched, substituted, or translated instead of the default "$_". The return value indicates the success of the operation. (If the right argument is an expression other than a search pattern, substitution, or translation, it is interpreted as a search pattern at run time. This is less efficient than an explicit search, since the pattern must be compiled every time the expression is evaluated.) The precedence of this operator is lower than unary minus and autoincrement/decrement, but higher than everything else.

!~ Just like =~ except the return value is negated.

x The repetition operator. Returns a string consisting of the left operand repeated the number of times specified by the right operand. In an array context, if the left operand is a list in parens, it repeats the list.

```
print '-' x 80;          # print row of dashes
print '-' x80;           # illegal, x80 is identifier

print "\t" x ($tab/8), ' ' x ($tab%8);     # tab over

@ones = (1) x 80;              # an array of 80 1's
@ones = (5) x @ones;           # set all elements to 5
```

x= The repetition assignment operator. Only works on scalars.

.. The range operator, which is really two different operators depending on the context. In an array context, returns an array of values counting (by ones) from the left value to the right value. This is useful for writing "for (1..10)" loops and for doing slice operations on arrays.

In a scalar context, .. returns a boolean value. The operator is bistable, like a flip-flop.. Each .. operator maintains its own boolean state. It is false as long as its left operand is false. Once the left operand is true, the range operator stays true until the right operand is true, AFTER which the range operator becomes false again. (It doesn't become false till the next time the range operator is evaluated. It can become false on the same evaluation it became true, but it still returns true once.) The right operand is not evaluated while the operator is in the "false" state, and the left operand is not evaluated while the operator is in the "true" state. The scalar .. operator is primarily intended for doing line number ranges after the fashion of *sed* or *awk*. The precedence is a little lower than || and &&. The value returned is either the null string for false, or a sequence number (beginning with 1) for true. The sequence number is reset for each range encountered. The final sequence number in a range has the string 'E0' appended to it, which doesn't affect its numeric value, but gives you something to search for if you want to exclude the endpoint. You can exclude the beginning point by waiting for the sequence number to be greater than 1. If either operand of scalar .. is static, that operand is implicitly compared to the $. variable, the current line number. Examples:

As a scalar operator:

```
if (101 .. 200) { print; }      # print 2nd hundred lines

next line if (1 .. /^$/);       # skip header lines

s/^/> / if (/^$/ .. eof());     # quote body
```

As an array operator:

```
for (101 .. 200) { print; }     # print $_ 100 times

@foo = @foo[$[ .. $#foo];  # an expensive no-op
@foo = @foo[$#foo-4 .. $#foo];     # slice last 5 items
```

-x A file test. This unary operator takes one argument, either a filename or a filehandle, and tests the associated file to see if something is true about it. If the argument is omitted, tests $_, except for -t, which tests *STDIN*. It returns 1 for true and '' for false, or

the undefined value if the file doesn't exist. Precedence is higher than logical and relational operators, but lower than arithmetic operators. The operator may be any of:

-r	File is readable by effective uid.
-w	File is writable by effective uid.
-x	File is executable by effective uid.
-o	File is owned by effective uid.
-R	File is readable by real uid.
-W	File is writable by real uid.
-X	File is executable by real uid.
-O	File is owned by real uid.
-e	File exists.
-z	File has zero size.
-s	File has non-zero size (returns size).
-f	File is a plain file.
-d	File is a directory.
-l	File is a symbolic link.
-p	File is a named pipe (FIFO).
-S	File is a socket.
-b	File is a block special file.
-c	File is a character special file.
-u	File has setuid bit set.
-g	File has setgid bit set.
-k	File has sticky bit set.
-t	Filehandle is opened to a tty.
-T	File is a text file.
-B	File is a binary file (opposite of -T).
-M	Age of file in days when script started.
-A	Same for access time.
-C	Same for inode change time.

The interpretation of the file permission operators -r, -R, -w, -W, -x and -X is based solely on the mode of the file and the uids and gids of the user. There may be other reasons you can't actually read, write or execute the file. Also note that, for the superuser, -r, -R, -w and -W always return 1, and -x and -X return 1 if any execute bit is set in the mode. Scripts run by the superuser may thus need to do a stat() in order to determine the actual mode of the file, or temporarily set the uid to something else.

Example:

```
while (<>) {
        chop;
        next unless -f $_;          # ignore specials
        ...
}
```

Note that -s/a/b/ does not do a negated substitution. Saying -exp($foo) still works as expected, however—only single letters following a minus are interpreted as file tests.

The -T and -B switches work as follows. The first block or so of the file is examined for odd characters such as strange control codes or metacharacters. If too many odd characters (>10%) are found, it's a -B file, otherwise it's a -T file. Also, any file containing null in the first block is considered a binary file. If -T or -B is used on a filehandle, the current stdio buffer is examined rather than the first block. Both -T and -B return TRUE on a null file, or a file at EOF when testing a filehandle.

If any of the file tests (or either stat operator) are given the special filehandle consisting of a solitary underline, then the stat structure of the previous file test (or stat operator) is used, saving a system call. (This doesn't work with -t, and you need to remember that lstat and -l will leave values in the stat structure for the symbolic link, not the real file.) Example:

```
print "Can do.\n" if -r $a || -w _ || -x _;

stat($filename);
print "Readable\n" if -r _;
print "Writable\n" if -w _;
print "Executable\n" if -x _;
print "Setuid\n" if -u _;
print "Setgid\n" if -g _;
print "Sticky\n" if -k _;
print "Text\n" if -T _;
print "Binary\n" if -B _;
```

Here is what C has that *perl* doesn't:

unary & Address-of operator.

unary * Dereference-address operator.

(TYPE) Type casting operator.

Like C, *perl* does a certain amount of expression evaluation at compile time, whenever it determines that all of the arguments to an operator are static and have no side effects. In particular, string concatenation happens at compile time between literals that don't do variable substitution. Backslash interpretation also happens at compile time. You can say

```
'Now is the time for all' . "\n" .
'good men to come to.'
```

and this all reduces to one string internally.

The autoincrement operator has a little extra built-in magic to it. If you increment a variable that is numeric, or that has ever been used in a numeric context, you get a normal increment. If, however, the variable has only been used in string contexts since it was set, and has a value that is not null and matches the pattern /^[a-zA-Z]*[0-9]*$/, the increment is done as a string, preserving each character within its range, with carry:

```
print ++($foo = '99'); # prints '100'
print ++($foo = 'a0'); # prints 'a1'
print ++($foo = 'Az'); # prints 'Ba'
print ++($foo = 'zz'); # prints 'aaa'
```

The autodecrement is not magical.

The range operator (in an array context) makes use of the magical autoincrement algorithm if the minimum and maximum are strings. You can say

```
@alphabet = ('A' .. 'Z');
```

to get all the letters of the alphabet, or

```
$hexdigit = (0 .. 9, 'a' .. 'f')[$num & 15];
```

to get a hexadecimal digit, or

```
@z2 = ('01' .. '31'); print @z2[$mday];
```

to get dates with leading zeros. (If the final value specified is not in the sequence that the magical increment would produce, the sequence goes until the next value would be longer than the final value specified.)

The || and && operators differ from C's in that, rather than returning 0 or 1, they return the last value evaluated. Thus, a portable way to find out the home directory might be:

```
$home = $ENV{'HOME'} || $ENV{'LOGDIR'} ||
    (getpwuid($<))[7] || die "You're homeless!\n";
```

Along with the literals and variables mentioned earlier, the operations in the following section can serve as terms in an expression. Some of these operations take a LIST as an argument. Such a list can consist of any combination of scalar arguments or array values; the array values will be included in the list as if each individual element were interpolated at that point in the list, forming a longer single-dimensional array value. Elements of the LIST should be separated by commas. If an operation is listed both with and without parentheses around its arguments, it means you can either use it as a unary operator or as a function call. To use it as a function call, the next token on the same line must be a left parenthesis. (There may be intervening white space.) Such a function then has highest precedence, as you would expect from a function. If any token other than a left parenthesis follows, then it is a unary operator, with a precedence depending only on whether it is a LIST operator or not. LIST operators have lowest precedence. All other unary operators have a precedence greater than relational operators but less than arithmetic operators. See the section on Precedence.

/PATTERN/

See m/PATTERN/.

?PATTERN?

This is just like the /pattern/ search, except that it matches only once between calls to the *reset* operator. This is a useful optimization when you only want to see the first occurrence of something in each file of a set of files, for instance. Only ?? patterns local to the current package are reset.

accept(NEWSOCKET,GENERICSOCKET)

Does the same thing that the accept system call does. Returns true if it succeeded, false otherwise. See example in section on Interprocess Communication.

alarm(SECONDS)

alarm SECONDS

Arranges to have a SIGALRM delivered to this process after the specified number of seconds (minus 1, actually) have elapsed. Thus, alarm(15) will cause a SIGALRM at some point more than 14 seconds in the future. Only one timer may be counting at once. Each call disables the previous timer, and an argument of 0 may be supplied to cancel the previous timer without starting a new one. The returned value is the amount of time remaining on the previous timer.

atan2(Y,X)

Returns the arctangent of Y/X in the range $-\pi$ to π.

bind(SOCKET,NAME)

Does the same thing that the bind system call does. Returns true if it succeeded, false otherwise. NAME should be a packed address of the proper type for the socket. See example in section on Interprocess Communication.

binmode(FILEHANDLE)

binmode FILEHANDLE

Arranges for the file to be read in "binary" mode in operating systems that distinguish between binary and text files. Files that are not read in binary mode have CR LF sequences translated to LF on input and LF translated to CR LF on output. Binmode has no effect under Unix. If FILEHANDLE is an expression, the value is taken as the name of the filehandle.

caller(EXPR)

caller Returns the context of the current subroutine call:

```
($package,$filename,$line) = caller;
```

With EXPR, returns some extra information that the debugger uses to print a stack trace. The value of EXPR indicates how many call frames to go back before the current one.

chdir(EXPR)

chdir EXPR

Changes the working directory to EXPR, if possible. If EXPR is omitted, changes to home directory. Returns 1 upon success, 0 otherwise. See example under *die*.

chmod(LIST)

chmod LIST

Changes the permissions of a list of files. The first element of the list must be the numerical mode. Returns the number of files successfully changed.

```
$cnt = chmod 0755, 'foo', 'bar';
chmod 0755, @executables;
```

chop(LIST)

chop(VARIABLE)

chop VARIABLE

chop Chops off the last character of a string and returns the character chopped. It's used primarily to remove the newline from the end of an input record, but is much more efficient than s/\n// because it neither scans nor copies the string. If VARIABLE is omitted, chops $_. Example:

```
while (<>) {
        chop;   # avoid \n on last field
        @array = split(/:/);
        ...
}
```

You can actually chop anything that's an lvalue, including an assignment:

```
chop($cwd = `pwd`);
chop($answer = <STDIN>);
```

If you chop a list, each element is chopped. Only the value of the last chop is returned.

chown(LIST)

chown LIST

Changes the owner (and group) of a list of files. The first two elements of the list must be the NUMERICAL uid and gid, in that order. Returns the number of files successfully changed.

```
$cnt = chown $uid, $gid, 'foo', 'bar';
chown $uid, $gid, @filenames;
```

Here's an example that looks up non-numeric uids in the passwd file:

```
print "User: ";
$user = <STDIN>;
chop($user);
print "Files: "
$pattern = <STDIN>;
chop($pattern);
open(pass, '/etc/passwd') || die "Can't open passwd: $!\n";
while (<pass>) {
        ($login,$pass,$uid,$gid) = split(/:/);
        $uid{$login} = $uid;
        $gid{$login} = $gid;
}
@ary = <${pattern}>;          # get filenames
if ($uid{$user} eq '') {
        die "$user not in passwd file";
}
else {
        chown $uid{$user}, $gid{$user}, @ary;
}
```

chroot(FILENAME)

chroot FILENAME

Does the same as the system call of that name. If you don't know what it does, don't worry about it. If FILENAME is omitted, does chroot to $_.

close(FILEHANDLE)

close FILEHANDLE

Closes the file or pipe associated with the file handle. You don't have to close FILEHANDLE if you are immediately going to do another open on it, since open will close it for you. (See *open*.) However, an explicit close on an input file resets the line counter ($.), while the implicit close done by *open* does not. Also, closing a pipe will wait for the process executing on the pipe to complete, in case you want to look at the output of the pipe afterwards. Closing a pipe explicitly also puts the status value of the command into $?. Example:

```
open(OUTPUT, '|sort >foo');  # pipe to sort
...      # print stuff to output
close OUTPUT;          # wait for sort to finish
open(INPUT, 'foo');    # get sort's results
```

FILEHANDLE may be an expression whose value gives the real filehandle name.

closedir(DIRHANDLE)

closedir DIRHANDLE

Closes a directory opened by opendir().

connect(SOCKET,NAME)

Does the same thing that the connect system call does. Returns true if it succeeded, false otherwise. NAME should be a package address of the proper type for the socket. See example in section on Interprocess Communication.

cos(EXPR)

cos EXPR

Returns the cosine of EXPR (expressed in radians). If EXPR is omitted takes cosine of $_.

crypt(PLAINTEXT,SALT)

Encrypts a string exactly like the crypt() function in the C library. Useful for checking the password file for lousy passwords. Only the guys wearing white hats should do this.

dbmclose(ASSOC_ARRAY)

dbmclose ASSOC_ARRAY

Breaks the binding between a dbm file and an associative array. The values remaining in the associative array are meaningless unless you happen to want to know what was in the cache for the dbm file. This function is only useful if you have ndbm.

dbmopen(ASSOC,DBNAME,MODE)

This binds a dbm or ndbm file to an associative array. ASSOC is the name of the associative array. (Unlike normal open, the first argument is NOT a filehandle, even though it looks like one). DBNAME is the name of the database (without the .dir or .pag extension). If the database does not exist, it is created with protection specified by MODE (as modified by the umask). If your system only supports the older dbm functions, you may only have one dbmopen in your program. If your system has neither dbm nor ndbm, calling dbmopen produces a fatal error.

Values assigned to the associative array prior to the dbmopen are lost. A certain number of values from the dbm file are cached in memory. By default this number is 64, but you can increase it by preallocating that number of garbage entries in the associative array before the dbmopen. You can flush the cache if necessary with the reset command.

If you don't have write access to the dbm file, you can only read associative array variables, not set them. If you want to test whether you can write, either use file tests or try setting a dummy array entry inside an eval, which will trap the error.

Note that functions such as keys() and values() may return huge array values when used on large dbm files. You may prefer to use the each() function to iterate over large dbm files. Example:

```
# print out history file offsets
dbmopen(HIST,'/usr/lib/news/history',0666);
while (($key,$val) = each %HIST) {
        print $key, ' = ', unpack('L',$val), "\n";
}
dbmclose(HIST);
```

defined(EXPR)

defined EXPR

Returns a boolean value saying whether the lvalue EXPR has a real value or not. Many operations return the undefined value under exceptional conditions, such as end of file, uninitialized variable, system error and such. This function allows you to distinguish between an undefined null string and a defined null string with operations that might return a real null string, in particular referencing elements of an array. You may also check to see if arrays or subroutines exist. Use on predefined variables is not guaranteed to produce intuitive results. Examples:

```
print if defined $switch{'D'};
print "$val\n" while defined($val = pop(@ary));
die "Can't readlink $sym: $!"
        unless defined($value = readlink $sym);
eval '@foo = ()' if defined(@foo);
die "No XYZ package defined" unless defined %_XYZ;
sub foo { defined &bar ? &bar(@_) : die "No bar"; }
```

See also undef.

delete $ASSOC{KEY}

Deletes the specified value from the specified associative array. Returns the deleted value, or the undefined value if nothing was deleted. Deleting from $ENV{} modifies the environment. Deleting from an array bound to a dbm file deletes the entry from the dbm file.

The following deletes all the values of an associative array:

```
foreach $key (keys %ARRAY) {
        delete $ARRAY{$key};
}
```

(But it would be faster to use the *reset* command. Saying undef %ARRAY is faster yet.)

die(LIST)

die LIST

Outside of an eval, prints the value of LIST to *STDERR* and exits with the current value of $! (errno). If $! is 0, exits with the value of ($? >> 8) (`command` status). If ($? >> 8) is 0, exits with 255. Inside an eval, the error message is stuffed into $@ and the eval is terminated with the undefined value.

Equivalent examples:

```
die "Can't cd to spool: $!\n" unless chdir '/usr/spool/news';

chdir '/usr/spool/news' || die "Can't cd to spool: $!\n"
```

If the value of EXPR does not end in a newline, the current script line number and input line number (if any) are also printed, and a newline is supplied. Hint: sometimes appending ", stopped" to your message will cause it to make better sense when the string "at foo line 123" is appended. Suppose you are running script "canasta".

```
die "/etc/games is no good";
die "/etc/games is no good, stopped";
```

produce, respectively

```
/etc/games is no good at canasta line 123.
/etc/games is no good, stopped at canasta line 123.
```

See also *exit*.

do BLOCK

Returns the value of the last command in the sequence of commands indicated by BLOCK. When modified by a loop modifier, executes the BLOCK once before testing the loop condition. (On other statements the loop modifiers test the conditional first.)

do SUBROUTINE (LIST)

Executes a SUBROUTINE declared by a *sub* declaration, and returns the value of the last expression evaluated in SUBROUTINE. If there is no subroutine by that name, produces a fatal error. (You may use the "defined" operator to determine if a subroutine exists.) If you pass arrays as part of LIST you may wish to pass the length of the array in front of each array. (See the section on subroutines later on.) SUBROUTINE may be a scalar variable, in which case the variable contains the name of the subroutine to execute. The parentheses are required to avoid confusion with the "do EXPR" form.

As an alternate form, you may call a subroutine by prefixing the name with an ampersand: &foo(@args). If you aren't passing any arguments, you don't have to use parentheses. If you omit the parentheses, no @_ array is passed to the subroutine. The & form is also used to specify subroutines to the defined and undef operators.

do EXPR

Uses the value of EXPR as a filename and executes the contents of the file as a *perl* script. Its primary use is to include subroutines from a *perl* subroutine library.

```
do 'stat.pl';
```

is just like

```
eval `cat stat.pl`;
```

except that it's more efficient, more concise, keeps track of the current filename for error messages, and searches all the **-I** libraries if the file isn't in the current directory (see also the @INC array in Predefined Names). It's the same, however, in that it does reparse the file every time you call it, so if you are going to use the file inside a loop you might prefer to use -P and #include, at the expense of a little more startup time. (The main problem with #include is that cpp doesn't grok # comments—a workaround is to use ";#" for standalone comments.) Note that the following are NOT equivalent:

```
do $foo;# eval a file
do $foo();      # call a subroutine
```

Note that inclusion of library routines is better done with the "require" operator.

dump LABEL

This causes an immediate core dump. Primarily this is so that you can use the undump program to turn your core dump into an executable binary after having initialized all your variables at the beginning of the program. When the new binary is executed it will begin by executing a "goto LABEL" (with all the restrictions that goto suffers). Think of it as a goto with an intervening core dump and reincarnation. If LABEL is omitted, restarts the program from the top. WARNING: any files opened at the time of the dump will NOT be open any more when the program is reincarnated, with possible resulting confusion on the part of perl. See also -u.

Example:

```
#!/usr/bin/perl
require 'getopt.pl';
require 'stat.pl';
%days = (
    'Sun',1,
    'Mon',2,
    'Tue',3,
    'Wed',4,
    'Thu',5,
    'Fri',6,
    'Sat',7);

dump QUICKSTART if $ARGV[0] eq '-d';

QUICKSTART:
    do Getopt('f');
```

On a CONVEX system, instead of an actual coredump, the GNUemacs unexec() function is called to create an executable file called *script*.perldump (where *script* is the name of your script), which can then be run directly.

each(ASSOC_ARRAY)

each ASSOC_ARRAY

Returns a 2 element array consisting of the key and value for the next value of an associative array, so that you can iterate over it. Entries are returned in an apparently random order. When the array is entirely read, a null array is returned (which when assigned produces a FALSE (0) value). The next call to each() after that will start iterating again. The iterator can be reset only by reading all the elements from the array. You must not modify the array while iterating over it. There is a single iterator for each associative array, shared by all each(), keys() and values() function calls in the program. The following prints out your environment like the printenv program, only in a different order:

```
while (($key,$value) = each %ENV) {
        print "$key=$value\n";
}
```

See also keys() and values().

eof(FILEHANDLE)

eof()

eof Returns 1 if the next read on FILEHANDLE will return end of file, or if FILEHANDLE is not open. FILEHANDLE may be an expression whose value gives the real filehandle name. (Note that this function actually reads a character and then ungetc's it, so it is not very useful in an interactive context.) An eof without an argument returns the eof status for the last file read. Empty parentheses () may be used to indicate the pseudo file formed of the files listed on the command line, i.e. eof() is reasonable to use inside a while (<>) loop to detect the end of only the last file. Use eof(ARGV) or eof without the parentheses to test EACH file in a while (<>) loop. Examples:

```
# insert dashes just before last line of last file
while (<>) {
        if (eof()) {
                print "----------\n";
        }
        print;
}

# reset line numbering on each input file
while (<>) {
        print "$.\t$_";
        if (eof) {          # Not eof().
                close(ARGV);
        }
}
```

eval(EXPR)

eval EXPR

EXPR is parsed and executed as if it were a little *perl* program. It is executed in the context of the current *perl* program, so that any variable settings, subroutine or format definitions remain afterwards. The value returned is the value of the last expression evaluated, just as with subroutines. If there is a syntax error or runtime error, or a die statement is executed, an undefined value is returned by eval, and $@ is set to the error message. If there was no error, $@ is guaranteed to be a null string. If EXPR is omitted, evaluates $_. The final semicolon, if any, may be omitted from the expression.

Note that, since eval traps otherwise-fatal errors, it is useful for determining whether a particular feature (such as dbmopen or symlink) is implemented. It is also Perl's exception trapping mechanism, where the die operator is used to raise exceptions.

exec(LIST)

exec LIST

If there is more than one argument in LIST, or if LIST is an array with more than one value, calls execvp() with the arguments in LIST. If there is only one scalar argument, the argument is checked for shell metacharacters. If there are any, the entire argument is passed to "/bin/sh -c" for parsing. If there are none, the argument is split into words and passed directly to execvp(), which is more efficient. Note: exec (and system) do not flush your output buffer, so you may need to set $| to avoid lost output. Examples:

```
exec '/bin/echo', 'Your arguments are: ', @ARGV;
exec "sort $outfile | uniq";
```

If you don't really want to execute the first argument, but want to lie to the program you are executing about its own name, you can specify the program you actually want to run by assigning that to a variable and putting the name of the variable in front of the LIST without a comma. (This always forces interpretation of the LIST as a multi-valued list, even if there is only a single scalar in the list.) Example:

```
$shell = '/bin/csh';
exec $shell '-sh';          # pretend it's a login shell
```

exit(EXPR)

exit EXPR

Evaluates EXPR and exits immediately with that value. Example:

```
$ans = <STDIN>;
exit 0 if $ans =~ /^[Xx]/;
```

See also *die*. If EXPR is omitted, exits with 0 status.

exp(EXPR)

exp EXPR

Returns *e* to the power of EXPR. If EXPR is omitted, gives exp($_).

fcntl(FILEHANDLE,FUNCTION,SCALAR)

Implements the fcntl(2) function. You'll probably have to say

```
require "fcntl.ph";         # probably /usr/local/lib/perl/fcntl.ph
```

first to get the correct function definitions. If fcntl.ph doesn't exist or doesn't have the correct definitions you'll have to roll your own, based on your C header files such as <sys/fcntl.h>. (There is a perl script called h2ph that comes with the perl kit which may help you in this.) Argument processing and value return works just like ioctl below. Note that fcntl will produce a fatal error if used on a machine that doesn't implement fcntl(2).

fileno(FILEHANDLE)

fileno FILEHANDLE

Returns the file descriptor for a filehandle. Useful for constructing bitmaps for select(). If FILEHANDLE is an expression, the value is taken as the name of the filehandle.

flock(FILEHANDLE,OPERATION)

Calls flock(2) on FILEHANDLE. See manual page for flock(2) for definition of OPERATION. Returns true for success, false on failure. Will produce a fatal error if used on a machine that doesn't implement flock(2). Here's a mailbox appender for BSD systems.

```
$LOCK_SH = 1;
$LOCK_EX = 2;
$LOCK_NB = 4;
$LOCK_UN = 8;

sub lock {
   flock(MBOX,$LOCK_EX);
   # and, in case someone appended
   # while we were waiting...
   seek(MBOX, 0, 2);
}

sub unlock {
   flock(MBOX,$LOCK_UN);
}

open(MBOX, ">>/usr/spool/mail/$ENV{'USER'}")
        || die "Can't open mailbox: $!";

do lock();
print MBOX $msg,"\n\n";
do unlock();
```

fork Does a fork() call. Returns the child pid to the parent process and 0 to the child process. Note: unflushed buffers remain unflushed in both processes, which means you may need to set $| to avoid duplicate output.

getc(FILEHANDLE)

getc FILEHANDLE

getc Returns the next character from the input file attached to FILEHANDLE, or a null string at EOF. If FILEHANDLE is omitted, reads from STDIN.

getlogin Returns the current login from /etc/utmp, if any. If null, use getpwuid.

```
$login = getlogin || (getpwuid($<))[0] || "Somebody";
```

getpeername(SOCKET)

Returns the packed sockaddr address of other end of the SOCKET connection.

```
# An internet sockaddr
$sockaddr = 'S n a4 x8';
$hersockaddr = getpeername(S);
($family, $port, $heraddr) = unpack($sockaddr,$hersockaddr);
```

getpgrp(PID)

getpgrp PID

Returns the current process group for the specified PID, 0 for the current process. Will produce a fatal error if used on a machine that doesn't implement getpgrp(2). If EXPR is omitted, returns process group of current process.

getppid Returns the process id of the parent process.

getpriority(WHICH,WHO)
Returns the current priority for a process, a process group, or a user. (See getpriority(2).) Will produce a fatal error if used on a machine that doesn't implement getpriority(2).

getpwnam(NAME)

getgrnam(NAME)

gethostbyname(NAME)

getnetbyname(NAME)

getprotobyname(NAME)

getpwuid(UID)

getgrgid(GID)

getservbyname(NAME,PROTO)

gethostbyaddr(ADDR,ADDRTYPE)

getnetbyaddr(ADDR,ADDRTYPE)

getprotobynumber(NUMBER)

getservbyport(PORT,PROTO)

getpwent

getgrent

gethostent

getnetent

getprotoent

getservent

setpwent

setgrent

sethostent(STAYOPEN)

setnetent(STAYOPEN)

setprotoent(STAYOPEN)

setservent(STAYOPEN)

endpwent

endgrent

endhostent

endnetent

endprotoent

endservent
These routines perform the same functions as their counterparts in the system library. The return values from the various get routines are as follows:

```
($name,$passwd,$uid,$gid,
   $quota,$comment,$gcos,$dir,$shell) = getpw...
($name,$passwd,$gid,$members) = getgr...
```

```
($name,$aliases,$addrtype,$length,@addrs) = gethost...
($name,$aliases,$addrtype,$net) = getnet...
($name,$aliases,$proto) = getproto...
($name,$aliases,$port,$proto) = getserv...
```

The $members value returned by getgr... is a space separated list of the login names of the members of the group.

The @addrs value returned by the gethost... functions is a list of the raw addresses returned by the corresponding system library call. In the Internet domain, each address is four bytes long and you can unpack it by saying something like:

```
($a,$b,$c,$d) = unpack('C4',$addr[0]);
```

getsockname(SOCKET)
Returns the packed sockaddr address of this end of the SOCKET connection.

```
# An internet sockaddr
$sockaddr = 'S n a4 x8';
$mysockaddr = getsockname(S);
($family, $port, $myaddr) = unpack($sockaddr,$mysockaddr);
```

getsockopt(SOCKET,LEVEL,OPTNAME)
Returns the socket option requested, or undefined if there is an error.

gmtime(EXPR)

gmtime EXPR
Converts a time as returned by the time function to a 9-element array with the time analyzed for the Greenwich timezone. Typically used as follows:

```
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = gmtime(time);
```

All array elements are numeric, and come straight out of a struct tm. In particular this means that $mon has the range 0..11 and $wday has the range 0..6. If EXPR is omitted, does gmtime(time).

goto LABEL
Finds the statement labeled with LABEL and resumes execution there. Currently you may only go to statements in the main body of the program that are not nested inside a do {} construct. This statement is not implemented very efficiently, and is here only to make the *sed-to-perl* translator easier. I may change its semantics at any time, consistent with support for translated *sed* scripts. Use it at your own risk. Better yet, don't use it at all.

grep(EXPR,LIST)
Evaluates EXPR for each element of LIST (locally setting $_ to each element) and returns the array value consisting of those elements for which the expression evaluated to true. In a scalar context, returns the number of times the expression was true.

```
@foo = grep(!/^#/, @bar);    # weed out comments
```

Note that, since $_ is a reference into the array value, it can be used to modify the elements of the array. While this is useful and supported, it can cause bizarre results if the LIST is not a named array.

hex(EXPR)

hex EXPR

Returns the decimal value of EXPR interpreted as an hex string. (To interpret strings that might start with 0 or 0x see oct().) If EXPR is omitted, uses $_.

index(STR,SUBSTR,POSITION)

index(STR,SUBSTR)

Returns the position of the first occurrence of SUBSTR in STR at or after POSITION. If POSITION is omitted, starts searching from the beginning of the string. The return value is based at 0, or whatever you've set the $[variable to. If the substring is not found, returns one less than the base, ordinarily -1.

int(EXPR)

int EXPR

Returns the integer portion of EXPR. If EXPR is omitted, uses $_.

ioctl(FILEHANDLE,FUNCTION,SCALAR)

Implements the ioctl(2) function. You'll probably have to say

```
require "ioctl.ph";        # probably /usr/local/lib/perl/ioctl.ph
```

first to get the correct function definitions. If ioctl.ph doesn't exist or doesn't have the correct definitions you'll have to roll your own, based on your C header files such as <sys/ioctl.h>. (There is a perl script called h2ph that comes with the perl kit which may help you in this.) SCALAR will be read and/or written depending on the FUNCTION—a pointer to the string value of SCALAR will be passed as the third argument of the actual ioctl call. (If SCALAR has no string value but does have a numeric value, that value will be passed rather than a pointer to the string value. To guarantee this to be true, add a 0 to the scalar before using it.) The pack() and unpack() functions are useful for manipulating the values of structures used by ioctl(). The following example sets the erase character to DEL.

```
require 'ioctl.ph';
$sgttyb_t = "cccs";              # 4 chars and a short
if (ioctl(STDIN,$TIOCGETP,$sgttyb)) {
        @ary = unpack($sgttyb_t,$sgttyb);
        $ary[2] = 127;
        $sgttyb = pack($sgttyb_t,@ary);
        ioctl(STDIN,$TIOCSETP,$sgttyb)
                || die "Can't ioctl: $!";
}
```

The return value of ioctl (and fcntl) is as follows:

if OS returns:	perl returns:
-1	undefined value
0	string "0 but true"
anything else	that number

Thus perl returns true on success and false on failure, yet you can still easily determine the actual value returned by the operating system:

```
($retval = ioctl(...)) || ($retval = -1);
printf "System returned %d\n", $retval;
```

join(EXPR,LIST)

join(EXPR,ARRAY)

Joins the separate strings of LIST or ARRAY into a single string with fields separated by the value of EXPR, and returns the string. Example:

```
$_ = join(':', $login,$passwd,$uid,$gid,$gcos,$home,$shell);
```

See *split*.

keys(ASSOC_ARRAY)

keys ASSOC_ARRAY

Returns a normal array consisting of all the keys of the named associative array. The keys are returned in an apparently random order, but it is the same order as either the values() or each() function produces (given that the associative array has not been modified). Here is yet another way to print your environment:

```
@keys = keys %ENV;
@values = values %ENV;
while ($#keys >= 0) {
        print pop(@keys), '=', pop(@values), "\n";
}
```

or how about sorted by key:

```
foreach $key (sort(keys %ENV)) {
        print $key, '=', $ENV{$key}, "\n";
}
```

kill(LIST)

kill LIST

Sends a signal to a list of processes. The first element of the list must be the signal to send. Returns the number of processes successfully signaled.

```
$cnt = kill 1, $child1, $child2;
kill 9, @goners;
```

If the signal is negative, kills process groups instead of processes. (On System V, a negative *process* number will also kill process groups, but that's not portable.) You may use a signal name in quotes.

last LABEL

last The *last* command is like the *break* statement in C (as used in loops); it immediately exits the loop in question. If the LABEL is omitted, the command refers to the innermost enclosing loop. The *continue* block, if any, is not executed:

```
line: while (<STDIN>) {
        last line if /^$/;          # exit when done with header
        ...
}
```

length(EXPR)

length EXPR

Returns the length in characters of the value of EXPR. If EXPR is omitted, returns length of $_.

link(OLDFILE,NEWFILE)

Creates a new filename linked to the old filename. Returns 1 for success, 0 otherwise.

listen(SOCKET,QUEUESIZE)

Does the same thing that the listen system call does. Returns true if it succeeded, false otherwise. See example in section on Interprocess Communication.

local(LIST)

Declares the listed variables to be local to the enclosing block, subroutine, eval or "do". All the listed elements must be legal lvalues. This operator works by saving the current values of those variables in LIST on a hidden stack and restoring them upon exiting the block, subroutine or eval. This means that called subroutines can also reference the local variable, but not the global one. The LIST may be assigned to if desired, which allows you to initialize your local variables. (If no initializer is given for a particular variable, it is created with an undefined value.) Commonly this is used to name the parameters to a subroutine. Examples:

```
sub RANGEVAL {
        local($min, $max, $thunk) = @_;
        local($result) = '';
        local($i);

        # Presumably $thunk makes reference to $i

        for ($i = $min; $i < $max; $i++) {
                $result .= eval $thunk;
        }

        $result;
}

if ($sw eq '-v') {
    # init local array with global array
    local(@ARGV) = @ARGV;
    unshift(@ARGV,'echo');
    system @ARGV;
}
# @ARGV restored

# temporarily add to digits associative array
if ($base12) {
        # (NOTE: not claiming this is efficient!)
        local(%digits) = (%digits,'t',10,'e',11);
        do parse_num();
}
```

Note that local() is a run-time command, and so gets executed every time through a loop, using up more stack storage each time until it's all released at once when the loop is exited.

localtime(EXPR)

localtime EXPR

Converts a time as returned by the time function to a 9-element array with the time analyzed for the local timezone. Typically used as follows:

($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime(time);

All array elements are numeric, and come straight out of a struct tm. In particular this means that $mon has the range 0..11 and $wday has the range 0..6. If EXPR is omitted, does localtime(time).

log(EXPR)

log EXPR

Returns logarithm (base *e*) of EXPR. If EXPR is omitted, returns log of $_.

lstat(FILEHANDLE)

lstat FILEHANDLE

lstat(EXPR)

lstat SCALARVARIABLE

Does the same thing as the stat() function, but stats a symbolic link instead of the file the symbolic link points to. If symbolic links are unimplemented on your system, a normal stat is done.

m/PATTERN/gio

/PATTERN/gio

Searches a string for a pattern match, and returns true (1) or false (''). If no string is specified via the =~ or !~ operator, the $_ string is searched. (The string specified with =~ need not be an lvalue—it may be the result of an expression evaluation, but remember the =~ binds rather tightly.) See also the section on regular expressions.

If / is the delimiter then the initial 'm' is optional. With the 'm' you can use any pair of non-alphanumeric characters as delimiters. This is particularly useful for matching Unix path names that contain '/'. If the final delimiter is followed by the optional letter 'i', the matching is done in a case-insensitive manner. PATTERN may contain references to scalar variables, which will be interpolated (and the pattern recompiled) every time the pattern search is evaluated. (Note that $) and $| may not be interpolated because they look like end-of-string tests.) If you want such a pattern to be compiled only once, add an "o" after the trailing delimiter. This avoids expensive run-time recompilations, and is useful when the value you are interpolating won't change over the life of the script. If the PATTERN evaluates to a null string, the most recent successful regular expression is used instead.

If used in a context that requires an array value, a pattern match returns an array consisting of the subexpressions matched by the parentheses in the pattern, i.e. ($1, $2, $3...). It does NOT actually set $1, $2, etc. in this case, nor does it set $+, $', $& or $'. If the match fails, a null array is returned. If the match succeeds, but there were no parentheses, an array value of (1) is returned.

Examples:

```
open(tty, '/dev/tty');
<tty> =~ /^y/i && do foo();      # do foo if desired

if (/Version: *([0-9.]*)/) { $version = $1; }

next if m#^/usr/spool/uucp#;

# poor man's grep
$arg = shift;
while (<>) {
        print if /$arg/o;       # compile only once
}

if (($F1, $F2, $Etc) = ($foo =~ /^(\S+)\s+(\S+)\s*(.*)/))
```

This last example splits $foo into the first two words and the remainder of the line, and assigns those three fields to $F1, $F2 and $Etc. The conditional is true if any variables were assigned, i.e. if the pattern matched.

The "g" modifier specifies global pattern matching—that is, matching as many times as possible within the string. How it behaves depends on the context. In an array context, it returns a list of all the substrings matched by all the parentheses in the regular expression. If there are no parentheses, it returns a list of all the matched strings, as if there were parentheses around the whole pattern. In a scalar context, it iterates through the string, returning TRUE each time it matches, and FALSE when it eventually runs out of matches. (In other words, it remembers where it left off last time and restarts the search at that point.) It presumes that you have not modified the string since the last match. Modifying the string between matches may result in undefined behavior. (You can actually get away with in-place modifications via substr() that do not change the length of the entire string. In general, however, you should be using s///g for such modifications.) Examples:

```
# array context
($one,$five,$fifteen) = (`uptime` =~ /(\d+\.\d+)/g);

# scalar context
$/ = 1; $* = 1;
while ($paragraph = <>) {
    while ($paragraph =~ /[a-z]['")]*[.!?]+['")]*\s/g) {
        $sentences++;
    }
}
print "$sentences\n";
```

mkdir(FILENAME,MODE)

Creates the directory specified by FILENAME, with permissions specified by MODE (as modified by umask). If it succeeds it returns 1, otherwise it returns 0 and sets $! (errno).

msgctl(ID,CMD,ARG)

Calls the System V IPC function msgctl. If CMD is &IPC_STAT, then ARG must be a variable which will hold the returned msqid_ds structure. Returns like ioctl: the undefined value for error, "0 but true" for zero, or the actual return value otherwise.

msgget(KEY,FLAGS)

Calls the System V IPC function msgget. Returns the message queue id, or the undefined value if there is an error.

msgsnd(ID,MSG,FLAGS)

Calls the System V IPC function msgsnd to send the message MSG to the message queue ID. MSG must begin with the long integer message type, which may be created with pack("L", $type). Returns true if successful, or false if there is an error.

msgrcv(ID,VAR,SIZE,TYPE,FLAGS)

Calls the System V IPC function msgrcv to receive a message from message queue ID into variable VAR with a maximum message size of SIZE. Note that if a message is received, the message type will be the first thing in VAR, and the maximum length of VAR is SIZE plus the size of the message type. Returns true if successful, or false if there is an error.

next LABEL

next The *next* command is like the *continue* statement in C; it starts the next iteration of the loop:

```
line: while (<STDIN>) {
        next line if /^#/;      # discard comments
        ...
}
```

Note that if there were a *continue* block on the above, it would get executed even on discarded lines. If the LABEL is omitted, the command refers to the innermost enclosing loop.

oct(EXPR)

oct EXPR

Returns the decimal value of EXPR interpreted as an octal string. (If EXPR happens to start off with 0x, interprets it as a hex string instead.) The following will handle decimal, octal and hex in the standard notation:

```
$val = oct($val) if $val =~ /^0/;
```

If EXPR is omitted, uses $_.

open(FILEHANDLE,EXPR)

open(FILEHANDLE)

open FILEHANDLE

Opens the file whose filename is given by EXPR, and associates it with FILEHANDLE. If FILEHANDLE is an expression, its value is used as the name of the real filehandle wanted. If EXPR is omitted, the scalar variable of the same name as the FILEHANDLE contains the filename. If the filename begins with "<" or nothing, the file is opened for input. If the filename begins with ">", the file is opened for output. If the filename begins with ">>", the file is opened for appending. (You can put a '+' in front of the '>' or '<' to indicate that you want both read and write access to the file.) If the filename begins with "|", the filename is interpreted as a command to which output is to be piped, and if the filename ends with a "|", the filename is interpreted as command which pipes input to us. (You may not have a command that pipes both in and out.) Opening '-' opens *STDIN* and opening '>-' opens *STDOUT*. Open returns non-zero upon success, the undefined value otherwise. If the open involved a pipe, the return value happens to be the pid of the subprocess. Examples:

```
$article = 100;
open article || die "Can't find article $article: $!\n";
while (<article>) {...

open(LOG, '>>/usr/spool/news/twitlog');     # (log is reserved)

open(article, "caesar <$article |");        # decrypt article

open(extract, "|sort >/tmp/Tmp$$");         # $$ is our process#

# process argument list of files along with any includes

foreach $file (@ARGV) {
        do process($file, 'fh00');# no pun intended
}

sub process {
        local($filename, $input) = @_;
        $input++;               # this is a string increment
        unless (open($input, $filename)) {
                print STDERR "Can't open $filename: $!\n";
                return;
        }
        while (<$input>) {              # note the use of indirection
                if (/^#include "(.*)"/) {
                        do process($1, $input);
                        next;
                }
                ...             # whatever
        }
}
```

You may also, in the Bourne shell tradition, specify an EXPR beginning with ">&", in which case the rest of the string is interpreted as the name of a filehandle (or file descriptor, if numeric) which is to be duped and opened. You may use & after >, >>, <, +>, +>> and +<. The mode you specify should match the mode of the original filehandle. Here is a script that saves, redirects, and restores *STDOUT* and *STDERR*:

```
#!/usr/bin/perl
open(SAVEOUT, ">&STDOUT");
open(SAVEERR, ">&STDERR");

open(STDOUT, ">foo.out") || die "Can't redirect stdout";
open(STDERR, ">&STDOUT") || die "Can't dup stdout";

select(STDERR); $| = 1;                 # make unbuffered
select(STDOUT); $| = 1;                 # make unbuffered

print STDOUT "stdout 1\n";    # this works for
print STDERR "stderr 1\n";    # subprocesses too

close(STDOUT);
close(STDERR);

open(STDOUT, ">&SAVEOUT");
open(STDERR, ">&SAVEERR");

print STDOUT "stdout 2\n";
print STDERR "stderr 2\n";
```

If you open a pipe on the command "-", i.e. either "|-" or "-|", then there is an implicit fork done, and the return value of open is the pid of the child within the parent process, and 0 within the child process. (Use defined($pid) to determine if the open was successful.) The filehandle behaves normally for the parent, but i/o to that filehandle is piped from/to the *STDOUT*/*STDIN* of the child process. In the child process the filehandle isn't opened—i/o happens from/to the new *STDOUT* or *STDIN*. Typically this is used like the normal piped open when you want to exercise more control over just how the pipe command gets executed, such as when you are running setuid, and don't want to have to scan shell commands for metacharacters. The following pairs are more or less equivalent:

```
open(FOO, "|tr '[a-z]' '[A-Z]'");
open(FOO, "|-") || exec 'tr', '[a-z]', '[A-Z]';

open(FOO, "cat -n '$file'|");
open(FOO, "-|") || exec 'cat', '-n', $file;
```

Explicitly closing any piped filehandle causes the parent process to wait for the child to finish, and returns the status value in $?. Note: on any operation which may do a fork, unflushed buffers remain unflushed in both processes, which means you may need to set $| to avoid duplicate output.

The filename that is passed to open will have leading and trailing whitespace deleted. In order to open a file with arbitrary weird characters in it, it's necessary to protect any leading and trailing whitespace thusly:

```
$file =~ s#^(\s)#./$1#;
open(FOO, "< $file\0");
```

opendir(DIRHANDLE,EXPR)

Opens a directory named EXPR for processing by readdir(), telldir(), seekdir(), rewinddir() and closedir(). Returns true if successful. DIRHANDLEs have their own namespace separate from FILEHANDLEs.

ord(EXPR)

ord EXPR

Returns the numeric ascii value of the first character of EXPR. If EXPR is omitted, uses $_.

pack(TEMPLATE,LIST)

Takes an array or list of values and packs it into a binary structure, returning the string containing the structure. The TEMPLATE is a sequence of characters that give the order and type of values, as follows:

A	An ascii string, will be space padded.
a	An ascii string, will be null padded.
c	A signed char value.
C	An unsigned char value.
s	A signed short value.
S	An unsigned short value.
i	A signed integer value.
I	An unsigned integer value.
l	A signed long value.
L	An unsigned long value.
n	A short in "network" order.
q	A signed long long value.
Q	An unsigned long long value.
N	A long in "network" order.
f	A single-precision float in the native format.
d	A double-precision float in the native format.
p	A pointer to a string.
x	A null byte.
X	Back up a byte.
@	Null fill to absolute position.
u	A uuencoded string.
b	A bit string (ascending bit order, like vec()).
B	A bit string (descending bit order).
h	A hex string (low nybble first).
H	A hex string (high nybble first).

Each letter may optionally be followed by a number which gives a repeat count. With all types except "a", "A", "b", "B", "h" and "H", the pack function will gobble up that many values from the LIST. A * for the repeat count means to use however many items are left. The "a" and "A" types gobble just one value, but pack it as a string of length count, padding with nulls or spaces as necessary. (When unpacking, "A" strips trailing spaces and nulls, but "a" does not.) Likewise, the "b" and "B" fields pack a string that many bits long. The "h" and "H" fields pack a string that many nybbles long. Real numbers (floats and doubles) are in the native machine format only; due to the multiplicity of floating formats around, and the lack of a standard "network" representation, no facility for interchange has been made. This means that packed floating point data written on one machine may not be readable on another - even if both use IEEE floating point arithmetic (as the endian-ness of the memory representation is not part of the IEEE spec). Note that perl uses doubles internally for all numeric calculation, and

converting from double -> float -> double will lose precision (i.e. unpack("f", pack("f", $foo)) will not in general equal $foo).
Examples:

```
$foo = pack("cccc",65,66,67,68);
# foo eq "ABCD"
$foo = pack("c4",65,66,67,68);
# same thing

$foo = pack("ccxxcc",65,66,67,68);
# foo eq "AB\0\0CD"

$foo = pack("s2",1,2);
# "\1\0\2\0" on little-endian
# "\0\1\0\2" on big-endian

$foo = pack("a4","abcd","x","y","z");
# "abcd"

$foo = pack("aaaa","abcd","x","y","z");
# "axyz"

$foo = pack("a14","abcdefg");
# "abcdefg\0\0\0\0\0\0\0"

$foo = pack("i9pl", gmtime);
# a real struct tm (on my system anyway)

sub bintodec {
    unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
}
```

The same template may generally also be used in the unpack function. The "q" and "Q" templates are only available on systems supporting the "long long" data type.

pipe(READHANDLE,WRITEHANDLE)

Opens a pair of connected pipes like the corresponding system call. Note that if you set up a loop of piped processes, deadlock can occur unless you are very careful. In addition, note that perl's pipes use stdio buffering, so you may need to set $| to flush your WRITEHANDLE after each command, depending on the application. [Requires version 3.0 patchlevel 9.]

pop(ARRAY)

pop ARRAY

Pops and returns the last value of the array, shortening the array by 1. Has the same effect as

```
$tmp = $ARRAY[$#ARRAY--];
```

If there are no elements in the array, returns the undefined value.

print(FILEHANDLE LIST)

print(LIST)

print FILEHANDLE LIST

print LIST

print Prints a string or a comma-separated list of strings. Returns non-zero if successful. FILEHANDLE may be a scalar variable name, in which case the variable contains the name of the filehandle, thus introducing one level of indirection. (NOTE: If FILEHANDLE is a variable and the next token is a term, it may be misinterpreted as an operator unless you interpose a + or put parens around the arguments.) If FILEHANDLE is omitted, prints by default to standard output (or to the last selected output channel—see select()). If LIST is also omitted, prints $_ to *STDOUT*. To set the default output channel to something other than *STDOUT* use the select operation. Note that, because print takes a LIST, anything in the LIST is evaluated in an array context, and any subroutine that you call will have one or more of its expressions evaluated in an array context. Also be careful not to follow the print keyword with a left parenthesis unless you want the corresponding right parenthesis to terminate the arguments to the print—interpose a + or put parens around all the arguments.

printf(FILEHANDLE LIST)

printf(LIST)

printf FILEHANDLE LIST

printf LIST

Equivalent to a "print FILEHANDLE sprintf(LIST)".

push(ARRAY,LIST)

Treats ARRAY (@ is optional) as a stack, and pushes the values of LIST onto the end of ARRAY. The length of ARRAY increases by the length of LIST. Has the same effect as

```
for $value (LIST) {
        $ARRAY[++$#ARRAY] = $value;
}
```

but is more efficient.

q/STRING/

qq/STRING/

qx/STRING/

These are not really functions, but simply syntactic sugar to let you avoid putting too many backslashes into quoted strings. The q operator is a generalized single quote, and the qq operator a generalized double quote. The qx operator is a generalized backquote. Any non-alphanumeric delimiter can be used in place of /, including newline. If the delimiter is an opening bracket or parenthesis, the final delimiter will be the corresponding closing bracket or parenthesis. (Embedded occurrences of the closing bracket need to be backslashed as usual.) Examples:

```
$foo = q!I said, "You said, 'She said it.'"!;
$bar = q('This is it.');
$today = qx{ date };
$_ .= qq
*** The previous line contains the naughty word "$&".\n
            if /(ibm|apple|awk)/;      # :-)
```

rand(EXPR)

rand EXPR

rand Returns a random fractional number between 0 and the value of EXPR. (EXPR should be positive.) If EXPR is omitted, returns a value between 0 and 1. See also srand().

read(FILEHANDLE,SCALAR,LENGTH,OFFSET)

read(FILEHANDLE,SCALAR,LENGTH)

Attempts to read LENGTH bytes of data into variable SCALAR from the specified FILEHANDLE. Returns the number of bytes actually read, or undef if there was an error. SCALAR will be grown or shrunk to the length actually read. An OFFSET may be specified to place the read data at some other place than the beginning of the string. This call is actually implemented in terms of stdio's fread call. To get a true read system call, see sysread.

readdir(DIRHANDLE)

readdir DIRHANDLE

Returns the next directory entry for a directory opened by opendir(). If used in an array context, returns all the rest of the entries in the directory. If there are no more entries, returns an undefined value in a scalar context or a null list in an array context.

readlink(EXPR)

readlink EXPR

Returns the value of a symbolic link, if symbolic links are implemented. If not, gives a fatal error. If there is some system error, returns the undefined value and sets $! (errno). If EXPR is omitted, uses $_.

recv(SOCKET,SCALAR,LEN,FLAGS)

Receives a message on a socket. Attempts to receive LENGTH bytes of data into variable SCALAR from the specified SOCKET filehandle. Returns the address of the sender, or the undefined value if there's an error. SCALAR will be grown or shrunk to the length actually read. Takes the same flags as the system call of the same name.

redo LABEL

redo The *redo* command restarts the loop block without evaluating the conditional again. The *continue* block, if any, is not executed. If the LABEL is omitted, the command refers to the innermost enclosing loop. This command is normally used by programs that want to lie to themselves about what was just input:

```
# a simpleminded Pascal comment stripper
# (warning: assumes no { or } in strings)
line: while (<STDIN>) {
        while (s|({.*}.*){.*}|$1 |) {}
        s|{.*}| |;
        if (s|{.*| |) {
                $front = $_;
                while (<STDIN>) {
                        if (/}/) {      # end of comment?
                                s|^|$front{|;
                                redo line;
                        }
                }
        }
        print;
}
```

rename(OLDNAME,NEWNAME)

Changes the name of a file. Returns 1 for success, 0 otherwise. Will not work across filesystem boundaries.

require(EXPR)

require EXPR

require Includes the library file specified by EXPR, or by $_ if EXPR is not supplied. Has semantics similar to the following subroutine:

```
sub require {
   local($filename) = @_;
   return 1 if $INC{$filename};
   local($realfilename,$result);
   ITER: {
        foreach $prefix (@INC) {
           $realfilename = "$prefix/$filename";
           if (-f $realfilename) {
                $result = do $realfilename;
                last ITER;
           }
        }
        die "Can't find $filename in \@INC";
   }
   die $@ if $@;
   die "$filename did not return true value" unless $result;
   $INC{$filename} = $realfilename;
   $result;
}
```

Note that the file will not be included twice under the same specified name.

reset(EXPR)

reset EXPR

reset Generally used in a *continue* block at the end of a loop to clear variables and reset ?? searches so that they work again. The expression is interpreted as a list of single characters (hyphens allowed for ranges). All variables and arrays beginning with one of those letters are reset to their pristine state. If the expression is omitted, one-match searches (?pattern?) are reset to match again. Only resets variables or searches in the current package. Always returns 1. Examples:

```
reset 'X';          # reset all X variables
reset 'a-z';        # reset lower case variables
reset;              # just reset ?? searches
```

Note: resetting "A-Z" is not recommended since you'll wipe out your ARGV and ENV arrays.

The use of reset on dbm associative arrays does not change the dbm file. (It does, however, flush any entries cached by perl, which may be useful if you are sharing the dbm file. Then again, maybe not.)

return LIST

Returns from a subroutine with the value specified. (Note that a subroutine can automatically return the value of the last expression evaluated. That's the preferred method—use of an explicit *return* is a bit slower.)

reverse(LIST)

reverse LIST

In an array context, returns an array value consisting of the elements of LIST in the opposite order. In a scalar context, returns a string value consisting of the bytes of the first element of LIST in the opposite order.

rewinddir(DIRHANDLE)

rewinddir DIRHANDLE

Sets the current position to the beginning of the directory for the readdir() routine on DIRHANDLE.

rindex(STR,SUBSTR,POSITION)

rindex(STR,SUBSTR)

Works just like index except that it returns the position of the LAST occurrence of SUBSTR in STR. If POSITION is specified, returns the last occurrence at or before that position.

rmdir(FILENAME)

rmdir FILENAME

Deletes the directory specified by FILENAME if it is empty. If it succeeds it returns 1, otherwise it returns 0 and sets $! (errno). If FILENAME is omitted, uses $_.

s/PATTERN/REPLACEMENT/gieo

Searches a string for a pattern, and if found, replaces that pattern with the replacement text and returns the number of substitutions made. Otherwise it returns false (0). The "g" is optional, and if present, indicates that all occurrences of the pattern are to be replaced. The "i" is also optional, and if present, indicates that matching is to be done in a case-insensitive manner. The "e" is likewise optional, and if present, indicates that the replacement string is to be evaluated as an expression rather than just as a double-quoted string. Any non-alphanumeric delimiter may replace the slashes; if single quotes

are used, no interpretation is done on the replacement string (the e modifier overrides this, however); if backquotes are used, the replacement string is a command to execute whose output will be used as the actual replacement text. If no string is specified via the =~ or !~ operator, the $_ string is searched and modified. (The string specified with =~ must be a scalar variable, an array element, or an assignment to one of those, i.e. an lvalue.) If the pattern contains a $ that looks like a variable rather than an end-of-string test, the variable will be interpolated into the pattern at run-time. If you only want the pattern compiled once the first time the variable is interpolated, add an "o" at the end. If the PATTERN evaluates to a null string, the most recent successful regular expression is used instead. See also the section on regular expressions. Examples:

```
s/\bgreen\b/mauve/g;                # don't change wintergreen

$path =~ s|/usr/bin|/usr/local/bin|;

s/Login: $foo/Login: $bar/; # run-time pattern

($foo = $bar) =~ s/bar/foo/;

$_ = 'abc123xyz';
s/\d+/$&*2/e;               # yields 'abc246xyz'
s/\d+/sprintf("%5d",$&)/e;  # yields 'abc  246xyz'
s/\w/$& x 2/eg;             # yields 'aabbcc  224466xxyyzz'

s/([^ ]*) *([^ ]*)/$2 $1/;  # reverse 1st two fields
```

(Note the use of $ instead of \ in the last example. See section on regular expressions.)

scalar(EXPR)

Forces EXPR to be interpreted in a scalar context and returns the value of EXPR.

seek(FILEHANDLE,POSITION,WHENCE)

Randomly positions the file pointer for FILEHANDLE, just like the fseek() call of stdio. FILEHANDLE may be an expression whose value gives the name of the filehandle. Returns 1 upon success, 0 otherwise.

seekdir(DIRHANDLE,POS)

Sets the current position for the readdir() routine on DIRHANDLE. POS must be a value returned by telldir(). Has the same caveats about possible directory compaction as the corresponding system library routine.

select(FILEHANDLE)

select Returns the currently selected filehandle. Sets the current default filehandle for output, if FILEHANDLE is supplied. This has two effects: first, a *write* or a *print* without a filehandle will default to this FILEHANDLE. Second, references to variables related to output will refer to this output channel. For example, if you have to set the top of form format for more than one output channel, you might do the following:

```
select(REPORT1);
$^ = 'report1_top';
select(REPORT2);
$^ = 'report2_top';
```

FILEHANDLE may be an expression whose value gives the name of the actual filehandle. Thus:

```
$oldfh = select(STDERR); $| = 1; select($oldfh);
```

select(RBITS,WBITS,EBITS,TIMEOUT)

This calls the select system call with the bitmasks specified, which can be constructed using fileno() and vec(), along these lines:

```
$rin = $win = $ein = '';
vec($rin,fileno(STDIN),1) = 1;
vec($win,fileno(STDOUT),1) = 1;
$ein = $rin | $win;
```

If you want to select on many filehandles you might wish to write a subroutine:

```
sub fhbits {
    local(@fhlist) = split(' ',$_[0]);
    local($bits);
    for (@fhlist) {
        vec($bits,fileno($_),1) = 1;
    }
    $bits;
}
$rin = &fhbits('STDIN TTY SOCK');
```

The usual idiom is:

```
($nfound,$timeleft) =
  select($rout=$rin, $wout=$win, $eout=$ein, $timeout);
```

or to block until something becomes ready:

```
$nfound = select($rout=$rin, $wout=$win, $eout=$ein, undef);
```

Any of the bitmasks can also be undef. The timeout, if specified, is in seconds, which may be fractional. NOTE: not all implementations are capable of returning the $timeleft. If not, they always return $timeleft equal to the supplied $timeout.

semctl(ID,SEMNUM,CMD,ARG)

Calls the System V IPC function semctl. If CMD is &IPC_STAT or &GETALL, then ARG must be a variable which will hold the returned semid_ds structure or semaphore value array. Returns like ioctl: the undefined value for error, "0 but true" for zero, or the actual return value otherwise.

semget(KEY,NSEMS,SIZE,FLAGS)

Calls the System V IPC function semget. Returns the semaphore id, or the undefined value if there is an error.

semop(KEY,OPSTRING)

Calls the System V IPC function semop to perform semaphore operations such as signaling and waiting. OPSTRING must be a packed array of semop structures. Each semop structure can be generated with 'pack("sss", $semnum, $semop, $semflag)'. The number of semaphore operations is implied by the length of OPSTRING. Returns true if successful, or false if there is an error. As an example, the following code waits on semaphore $semnum of semaphore id $semid:

```
$semop = pack("sss", $semnum, -1, 0);
```

```
die "Semaphore trouble: $!\n" unless semop($semid, $semop);
```

To signal the semaphore, replace "-1" with "1".

send(SOCKET,MSG,FLAGS,TO)

send(SOCKET,MSG,FLAGS)

Sends a message on a socket. Takes the same flags as the system call of the same name. On unconnected sockets you must specify a destination to send TO. Returns the number of characters sent, or the undefined value if there is an error.

setpgrp(PID,PGRP)

Sets the current process group for the specified PID, 0 for the current process. Will produce a fatal error if used on a machine that doesn't implement setpgrp(2).

setpriority(WHICH,WHO,PRIORITY)

Sets the current priority for a process, a process group, or a user. (See setpriority(2).) Will produce a fatal error if used on a machine that doesn't implement setpriority(2).

setsockopt(SOCKET,LEVEL,OPTNAME,OPTVAL)

Sets the socket option requested. Returns undefined if there is an error. OPTVAL may be specified as undef if you don't want to pass an argument.

shift(ARRAY)

shift ARRAY

shift Shifts the first value of the array off and returns it, shortening the array by 1 and moving everything down. If there are no elements in the array, returns the undefined value. If ARRAY is omitted, shifts the @ARGV array in the main program, and the @_ array in subroutines. (This is determined lexically.) See also unshift(), push() and pop(). Shift() and unshift() do the same thing to the left end of an array that push() and pop() do to the right end.

shmctl(ID,CMD,ARG)

Calls the System V IPC function shmctl. If CMD is &IPC_STAT, then ARG must be a variable which will hold the returned shmid_ds structure. Returns like ioctl: the undefined value for error, "0 but true" for zero, or the actual return value otherwise.

shmget(KEY,SIZE,FLAGS)

Calls the System V IPC function shmget. Returns the shared memory segment id, or the undefined value if there is an error.

shmread(ID,VAR,POS,SIZE)

shmwrite(ID,STRING,POS,SIZE)

Reads or writes the System V shared memory segment ID starting at position POS for size SIZE by attaching to it, copying in/out, and detaching from it. When reading, VAR must be a variable which will hold the data read. When writing, if STRING is too long, only SIZE bytes are used; if STRING is too short, nulls are written to fill out SIZE bytes. Return true if successful, or false if there is an error.

shutdown(SOCKET,HOW)

Shuts down a socket connection in the manner indicated by HOW, which has the same interpretation as in the system call of the same name.

sin(EXPR)

sin EXPR

Returns the sine of EXPR (expressed in radians). If EXPR is omitted, returns sine of $_.

sleep(EXPR)

sleep EXPR

sleep Causes the script to sleep for EXPR seconds, or forever if no EXPR. May be interrupted by sending the process a SIGALARM. Returns the number of seconds actually slept.

socket(SOCKET,DOMAIN,TYPE,PROTOCOL)

Opens a socket of the specified kind and attaches it to filehandle SOCKET. DOMAIN, TYPE and PROTOCOL are specified the same as for the system call of the same name. You may need to run h2ph on sys/socket.h to get the proper values handy in a perl library file. Return true if successful. See the example in the section on Interprocess Communication.

socketpair(SOCKET1,SOCKET2,DOMAIN,TYPE,PROTOCOL)

Creates an unnamed pair of sockets in the specified domain, of the specified type. DOMAIN, TYPE and PROTOCOL are specified the same as for the system call of the same name. If unimplemented, yields a fatal error. Return true if successful.

sort(SUBROUTINE LIST)

sort(LIST)

sort SUBROUTINE LIST

sort LIST

Sorts the LIST and returns the sorted array value. Nonexistent values of arrays are stripped out. If SUBROUTINE is omitted, sorts in standard string comparison order. If SUBROUTINE is specified, gives the name of a subroutine that returns an integer less than, equal to, or greater than 0, depending on how the elements of the array are to be ordered. (The <=> and cmp operators are extremely useful in such routines.) In the interests of efficiency the normal calling code for subroutines is bypassed, with the following effects: the subroutine may not be a recursive subroutine, and the two elements to be compared are passed into the subroutine not via @_ but as $a and $b (see example below). They are passed by reference so don't modify $a and $b. SUBROUTINE may be a scalar variable name, in which case the value provides the name of the subroutine to use. Examples:

```
sub byage {
   $age{$a} <=> $age{$b};     # presuming integers
}
@sortedclass = sort byage @class;

sub reverse { $b cmp $a; }
@harry = ('dog','cat','x','Cain','Abel');
@george = ('gone','chased','yz','Punished','Axed');
print sort @harry;
        # prints AbelCaincatdogx
print sort reverse @harry;
        # prints xdogcatCainAbel
print sort @george, 'to', @harry;
        # prints AbelAxedCainPunishedcatchaseddoggonetoxyz
```

splice(ARRAY,OFFSET,LENGTH,LIST)

splice(ARRAY,OFFSET,LENGTH)

splice(ARRAY,OFFSET)

Removes the elements designated by OFFSET and LENGTH from an array, and replaces them with the elements of LIST, if any. Returns the elements removed from the array. The array grows or shrinks as necessary. If LENGTH is omitted, removes everything from OFFSET onward. The following equivalencies hold (assuming $[== 0):

```
push(@a,$x,$y)          splice(@a,$#a+1,0,$x,$y)
pop(@a)                 splice(@a,-1)
shift(@a)               splice(@a,0,1)
unshift(@a,$x,$y)       splice(@a,0,0,$x,$y)
$a[$x] = $y             splice(@a,$x,1,$y);
```

Example, assuming array lengths are passed before arrays:

```
sub aeq {       # compare two array values
        local(@a) = splice(@_,0,shift);
        local(@b) = splice(@_,0,shift);
        return 0 unless @a === @b;     # same len?
        while (@a) {
           return 0 if pop(@a) ne pop(@b);
        }
        return 1;
}
if (&aeq($len,@foo[1..$len],0+@bar,@bar)) { ... }
```

split(/PATTERN/,EXPR,LIMIT)

split(/PATTERN/,EXPR)

split(/PATTERN/)

split Splits a string into an array of strings, and returns it. (If not in an array context, returns the number of fields found and splits into the @_ array. (In an array context, you can force the split into @_ by using ?? as the pattern delimiters, but it still returns the array value.)) If EXPR is omitted, splits the $_ string. If PATTERN is also omitted, splits on whitespace (/[\t\n]+/). Anything matching PATTERN is taken to be a delimiter separating the fields. (Note that the delimiter may be longer than one character.) If LIMIT is specified, splits into no more than that many fields (though it may split into fewer). If LIMIT is unspecified, trailing null fields are stripped (which potential users of pop() would do well to remember). A pattern matching the null string (not to be confused with a null pattern //, which is just one member of the set of patterns matching a null string) will split the value of EXPR into separate characters at each point it matches that way. For example:

```
print join(':', split(/ */, 'hi there'));
```

produces the output 'h:i:t:h:e:r:e'.

The LIMIT parameter can be used to partially split a line

```
($login, $passwd, $remainder) = split(/:/, $_, 3);
```

(When assigning to a list, if LIMIT is omitted, perl supplies a LIMIT one larger than the number of variables in the list, to avoid unnecessary work. For the list above LIMIT would have been 4 by default. In time critical applications it behooves you not to split into more fields than you really need.)

If the PATTERN contains parentheses, additional array elements are created from each matching substring in the delimiter.

```
split(/([,-])/,"1-10,20");
```

produces the array value

```
(1,'-',10,',',20)
```

The pattern /PATTERN/ may be replaced with an expression to specify patterns that vary at runtime. (To do runtime compilation only once, use /$variable/o.) As a special case, specifying a space (' ') will split on white space just as split with no arguments does, but leading white space does NOT produce a null first field. Thus, split(' ') can be used to emulate *awk*'s default behavior, whereas split(/ /) will give you as many null initial fields as there are leading spaces.

Example:

```
open(passwd, '/etc/passwd');
while (<passwd>) {
        ($login, $passwd, $uid, $gid, $gcos, $home, $shell) = split(/:/);
        ...
}
```

(Note that $shell above will still have a newline on it. See chop().) See also *join*.

sprintf(FORMAT,LIST)

Returns a string formatted by the usual printf conventions. The * character is not supported.

sqrt(EXPR)

sqrt EXPR

Return the square root of EXPR. If EXPR is omitted, returns square root of $_.

srand(EXPR)

srand EXPR

Sets the random number seed for the *rand* operator. If EXPR is omitted, does srand(time).

stat(FILEHANDLE)

stat FILEHANDLE

stat(EXPR)

stat SCALARVARIABLE

Returns a 13-element array giving the statistics for a file, either the file opened via FILEHANDLE, or named by EXPR. Typically used as follows:

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
  $atime,$mtime,$ctime,$blksize,$blocks)
    = stat($filename);
```

If stat is passed the special filehandle consisting of an underline, no stat is done, but the current contents of the stat structure from the last stat or filetest are returned.

Example:

```
if (-x $file && (($d) = stat(_)) && $d < 0) {
        print "$file is executable NFS file\n";
}
```

(This only works on machines for which the device number is negative under NFS.)

study(SCALAR)

study SCALAR

study Takes extra time to study SCALAR ($_ if unspecified) in anticipation of doing many pattern matches on the string before it is next modified. This may or may not save time, depending on the nature and number of patterns you are searching on, and on the distribution of character frequencies in the string to be searched—you probably want to compare runtimes with and without it to see which runs faster. Those loops which scan for many short constant strings (including the constant parts of more complex patterns) will benefit most. You may have only one study active at a time—if you study a different scalar the first is "unstudied". (The way study works is this: a linked list of every character in the string to be searched is made, so we know, for example, where all the 'k' characters are. From each search string, the rarest character is selected, based on some static frequency tables constructed from some C programs and English text. Only those places that contain this "rarest" character are examined.)

For example, here is a loop which inserts index producing entries before any line containing a certain pattern:

```
while (<>) {
        study;
        print ".IX foo\n" if /\bfoo\b/;
        print ".IX bar\n" if /\bbar\b/;
        print ".IX blurfl\n" if /\bblurfl\b/;
        ...
        print;
}
```

In searching for /\bfoo\b/, only those locations in $_ that contain 'f' will be looked at, because 'f' is rarer than 'o'. In general, this is a big win except in pathological cases. The only question is whether it saves you more time than it took to build the linked list in the first place.

Note that if you have to look for strings that you don't know till runtime, you can build an entire loop as a string and eval that to avoid recompiling all your patterns all the time. Together with undefining $/ to input entire files as one record, this can be very fast, often faster than specialized programs like fgrep. The following scans a list of files (@files) for a list of words (@words), and prints out the names of those files that contain a match:

```
$search = 'while (<>) { study;';
foreach $word (@words) {
    $search .= "++\$seen{\$ARGV} if /\b$word\b/;\n";
}
$search .= "}";
@ARGV = @files;
undef $/;
eval $search;                   # this screams
$/ = "\n";                      # put back to normal input delim
foreach $file (sort keys(%seen)) {
    print $file, "\n";
}
```

substr(EXPR,OFFSET,LEN)

substr(EXPR,OFFSET)

Extracts a substring out of EXPR and returns it. First character is at offset 0, or whatever you've set $[to. If OFFSET is negative, starts that far from the end of the string. If LEN is omitted, returns everything to the end of the string. You can use the substr() function as an lvalue, in which case EXPR must be an lvalue. If you assign something shorter than LEN, the string will shrink, and if you assign something longer than LEN, the string will grow to accommodate it. To keep the string the same length you may need to pad or chop your value using sprintf().

symlink(OLDFILE,NEWFILE)

Creates a new filename symbolically linked to the old filename. Returns 1 for success, 0 otherwise. On systems that don't support symbolic links, produces a fatal error at run time. To check for that, use eval:

```
$symlink_exists = (eval 'symlink("","");', $@ eq '');
```

syscall(LIST)

syscall LIST

Calls the system call specified as the first element of the list, passing the remaining elements as arguments to the system call. If unimplemented, produces a fatal error. The arguments are interpreted as follows: if a given argument is numeric, the argument is passed as an int. If not, the pointer to the string value is passed. You are responsible to make sure a string is pre-extended long enough to receive any result that might be written into a string. If your integer arguments are not literals and have never been interpreted in a numeric context, you may need to add 0 to them to force them to look like numbers.

```
require 'syscall.ph';               # may need to run h2ph
syscall(&SYS_write, fileno(STDOUT), "hi there\n", 9);
```

sysread(FILEHANDLE,SCALAR,LENGTH,OFFSET)

sysread(FILEHANDLE,SCALAR,LENGTH)

Attempts to read LENGTH bytes of data into variable SCALAR from the specified FILEHANDLE, using the system call read(2). It bypasses stdio, so mixing this with other kinds of reads may cause confusion. Returns the number of bytes actually read, or undef if there was an error. SCALAR will be grown or shrunk to the length actually read. An OFFSET may be specified to place the read data at some other place than the beginning of the string.

system(LIST)

system LIST

Does exactly the same thing as "exec LIST" except that a fork is done first, and the parent process waits for the child process to complete. Note that argument processing varies depending on the number of arguments. The return value is the exit status of the program as returned by the wait() call. To get the actual exit value divide by 256. See also *exec*.

syswrite(FILEHANDLE,SCALAR,LENGTH,OFFSET)

syswrite(FILEHANDLE,SCALAR,LENGTH)

Attempts to write LENGTH bytes of data from variable SCALAR to the specified FILEHANDLE, using the system call write(2). It bypasses stdio, so mixing this with prints may cause confusion. Returns the number of bytes actually written, or undef if there was an error. An OFFSET may be specified to place the read data at some other place than the beginning of the string.

tell(FILEHANDLE)

tell FILEHANDLE

tell Returns the current file position for FILEHANDLE. FILEHANDLE may be an expression whose value gives the name of the actual filehandle. If FILEHANDLE is omitted, assumes the file last read.

telldir(DIRHANDLE)

telldir DIRHANDLE

Returns the current position of the readdir() routines on DIRHANDLE. Value may be given to seekdir() to access a particular location in a directory. Has the same caveats about possible directory compaction as the corresponding system library routine.

time Returns the number of non-leap seconds since 00:00:00 UTC, January 1, 1970. Suitable for feeding to gmtime() and localtime().

times Returns a four-element array giving the user and system times, in seconds, for this process and the children of this process.

($user,$system,$cuser,$csystem) = times;

tr/SEARCHLIST/REPLACEMENTLIST/cds

y/SEARCHLIST/REPLACEMENTLIST/cds

Translates all occurrences of the characters found in the search list with the corresponding character in the replacement list. It returns the number of characters replaced or deleted. If no string is specified via the =~ or !~ operator, the $_ string is translated. (The string specified with =~ must be a scalar variable, an array element, or an assignment to one of those, i.e. an lvalue.) For *sed* devotees, *y* is provided as a synonym for *tr*.

If the c modifier is specified, the SEARCHLIST character set is complemented. If the d modifier is specified, any characters specified by SEARCHLIST that are not found in REPLACEMENTLIST are deleted. (Note that this is slightly more flexible than the behavior of some *tr* programs, which delete anything they find in the SEARCHLIST, period.) If the s modifier is specified, sequences of characters that were translated to the same character are squashed down to 1 instance of the character.

If the d modifier was used, the REPLACEMENTLIST is always interpreted exactly as specified. Otherwise, if the REPLACEMENTLIST is shorter than the SEARCHLIST, the final character is replicated till it is long enough. If the REPLACEMENTLIST is null, the SEARCHLIST is replicated. This latter is useful for counting characters in a

class, or for squashing character sequences in a class.

Examples:

```
$ARGV[1] =~ y/A-Z/a-z/;                 # canonicalize to lower case

$cnt = tr/*/*/;                         # count the stars in $_

$cnt = tr/0-9//;                        # count the digits in $_

tr/a-zA-Z//s;                           # bookkeeper -> bokeper

($HOST = $host) =~ tr/a-z/A-Z/;

y/a-zA-Z/ /cs;                          # change non-alphas to single space

tr/\200-\377/\0-\177/;                  # delete 8th bit
```

truncate(FILEHANDLE,LENGTH)

truncate(EXPR,LENGTH)

Truncates the file opened on FILEHANDLE, or named by EXPR, to the specified length. Produces a fatal error if truncate isn't implemented on your system.

umask(EXPR)

umask EXPR

umask Sets the umask for the process and returns the old one. If EXPR is omitted, merely returns current umask.

undef(EXPR)

undef EXPR

undef Undefines the value of EXPR, which must be an lvalue. Use only on a scalar value, an entire array, or a subroutine name (using &). (Undef will probably not do what you expect on most predefined variables or dbm array values.) Always returns the undefined value. You can omit the EXPR, in which case nothing is undefined, but you still get an undefined value that you could, for instance, return from a subroutine. Examples:

```
undef $foo;
undef $bar{'blurfl'};
undef @ary;
undef %assoc;
undef &mysub;
return (wantarray ? () : undef) if $they_blew_it;
```

unlink(LIST)

unlink LIST

Deletes a list of files. Returns the number of files successfully deleted.

```
$cnt = unlink 'a', 'b', 'c';
unlink @goners;
unlink <*.bak>;
```

Note: unlink will not delete directories unless you are superuser and the -U flag is

supplied to *perl*. Even if these conditions are met, be warned that unlinking a directory can inflict damage on your filesystem. Use rmdir instead.

unpack(TEMPLATE,EXPR)

Unpack does the reverse of pack: it takes a string representing a structure and expands it out into an array value, returning the array value. (In a scalar context, it merely returns the first value produced.) The TEMPLATE has the same format as in the pack function. Here's a subroutine that does substring:

```
sub substr {
        local($what,$where,$howmuch) = @_;
        unpack("x$where a$howmuch", $what);
}
```

and then there's

```
sub ord { unpack("c",$_[0]); }
```

In addition, you may prefix a field with a %<number> to indicate that you want a <number>-bit checksum of the items instead of the items themselves. Default is a 16-bit checksum. For example, the following computes the same number as the System V sum program:

```
while (<>) {
   $checksum += unpack("%16C*", $_);
}
$checksum %= 65536;
```

unshift(ARRAY,LIST)

Does the opposite of a *shift*. Or the opposite of a *push*, depending on how you look at it. Prepends list to the front of the array, and returns the number of elements in the new array.

```
unshift(ARGV, '-e') unless $ARGV[0] =~ /^-/;
```

utime(LIST)

utime LIST

Changes the access and modification times on each file of a list of files. The first two elements of the list must be the NUMERICAL access and modification times, in that order. Returns the number of files successfully changed. The inode modification time of each file is set to the current time. Example of a "touch" command:

```
#!/usr/bin/perl
$now = time;
utime $now, $now, @ARGV;
```

values(ASSOC_ARRAY)

values ASSOC_ARRAY

Returns a normal array consisting of all the values of the named associative array. The values are returned in an apparently random order, but it is the same order as either the keys() or each() function would produce on the same array. See also keys() and each().

vec(EXPR,OFFSET,BITS)

Treats a string as a vector of unsigned integers, and returns the value of the bitfield specified. May also be assigned to. BITS must be a power of two from 1 to 32.

Vectors created with vec() can also be manipulated with the logical operators |, & and ^, which will assume a bit vector operation is desired when both operands are strings. This interpretation is not enabled unless there is at least one vec() in your program, to protect older programs.

To transform a bit vector into a string or array of 0's and 1's, use these:

```
$bits = unpack("b*", $vector);
@bits = split(//, unpack("b*", $vector));
```

If you know the exact length in bits, it can be used in place of the *.

wait Waits for a child process to terminate and returns the pid of the deceased process, or -1 if there are no child processes. The status is returned in $?.

waitpid(PID,FLAGS)

Waits for a particular child process to terminate and returns the pid of the deceased process, or -1 if there is no such child process. The status is returned in $?. If you say

```
require "sys/wait.h";
...
waitpid(-1,&WNOHANG);
```

then you can do a non-blocking wait for any process. Non-blocking wait is only available on machines supporting either the *waitpid (2)* or *wait4 (2)* system calls. However, waiting for a particular pid with FLAGS of 0 is implemented everywhere. (Perl emulates the system call by remembering the status values of processes that have exited but have not been harvested by the Perl script yet.)

wantarray

Returns true if the context of the currently executing subroutine is looking for an array value. Returns false if the context is looking for a scalar.

```
return wantarray ? () : undef;
```

warn(LIST)

warn LIST

Produces a message on STDERR just like "die", but doesn't exit.

write(FILEHANDLE)

write(EXPR)

write Writes a formatted record (possibly multi-line) to the specified file, using the format associated with that file. By default the format for a file is the one having the same name is the filehandle, but the format for the current output channel (see *select*) may be set explicitly by assigning the name of the format to the $~ variable.

Top of form processing is handled automatically: if there is insufficient room on the current page for the formatted record, the page is advanced by writing a form feed, a special top-of-page format is used to format the new page header, and then the record is written. By default the top-of-page format is "top", but it may be set to the format of your choice by assigning the name to the $^ variable. The number of lines remaining on the current page is in variable $-, which can be set to 0 to force a new page.

If FILEHANDLE is unspecified, output goes to the current default output channel, which starts out as *STDOUT* but may be changed by the *select* operator. If the FILEHANDLE is an EXPR, then the expression is evaluated and the resulting string is used to look up the name of the FILEHANDLE at run time. For more on formats, see the section on formats later on.

Note that write is NOT the opposite of read.

Precedence

Perl operators have the following associativity and precedence:

nonassoc	print printf exec system sort reverse chmod chown kill unlink utime die return
left	,
right	= += -= *= etc.
right	?:
nonassoc	..
left	||
left	&&
left	| ^
left	&
nonassoc	== != <=> eq ne cmp
nonassoc	< > <= >= lt gt le ge
nonassoc	chdir exit eval reset sleep rand umask
nonassoc	-r -w -x etc.
left	<< >>
left	+ - .
left	* / % x
left	=~ !~
right	! ~ and unary minus
right	**
nonassoc	++ --
left	'('

As mentioned earlier, if any list operator (print, etc.) or any unary operator (chdir, etc.) is followed by a left parenthesis as the next token on the same line, the operator and arguments within parentheses are taken to be of highest precedence, just like a normal function call. Examples:

```
chdir $foo || die;          # (chdir $foo) || die
chdir($foo) || die;         # (chdir $foo) || die
chdir ($foo) || die;        # (chdir $foo) || die
chdir +($foo) || die;       # (chdir $foo) || die
```

but, because * is higher precedence than ||:

```
chdir $foo * 20;            # chdir ($foo * 20)
chdir($foo) * 20;           # (chdir $foo) * 20
chdir ($foo) * 20;          # (chdir $foo) * 20
chdir +($foo) * 20;         # chdir ($foo * 20)

rand 10 * 20;               # rand (10 * 20)
rand(10) * 20;              # (rand 10) * 20
rand (10) * 20;             # (rand 10) * 20
rand +(10) * 20;            # rand (10 * 20)
```

In the absence of parentheses, the precedence of list operators such as print, sort or chmod is either very high or very low depending on whether you look at the left side of operator or the right side of it. For example, in

```
@ary = (1, 3, sort 4, 2);
print @ary;             # prints 1324
```

the commas on the right of the sort are evaluated before the sort, but the commas on the left are evaluated after. In other words, list operators tend to gobble up all the arguments that follow them, and then act like a simple term with regard to the preceding expression. Note that you have to be careful with parens:

```
# These evaluate exit before doing the print:
print($foo, exit);      # Obviously not what you want.
print $foo, exit; # Nor is this.

# These do the print before evaluating exit:
(print $foo), exit;     # This is what you want.
print($foo), exit;      # Or this.
print ($foo), exit;     # Or even this.
```

Also note that

```
print ($foo & 255) + 1, "\n";
```

probably doesn't do what you expect at first glance.

Subroutines

A subroutine may be declared as follows:

```
sub NAME BLOCK
```

Any arguments passed to the routine come in as array @_, that is ($_[0], $_[1], ...). The array @_ is a local array, but its values are references to the actual scalar parameters. The return value of the subroutine is the value of the last expression evaluated, and can be either an array value or a scalar value. Alternately, a return statement may be used to specify the returned value and exit the subroutine. To create local variables see the *local* operator.

A subroutine is called using the *do* operator or the & operator.

Example:

```
sub MAX {
        local($max) = pop(@_);
        foreach $foo (@_) {
                $max = $foo if $max < $foo;
        }
        $max;
}

...
$bestday = &MAX($mon,$tue,$wed,$thu,$fri);
```

Example:

```
# get a line, combining continuation lines
#  that start with whitespace
sub get_line {
        $thisline = $lookahead;
        line: while ($lookahead = <STDIN>) {
                if ($lookahead =~ /^[ \t]/) {
                        $thisline .= $lookahead;
                }
                else {
                        last line;
                }
        }
        $thisline;
}

$lookahead = <STDIN>;       # get first line
while ($_ = do get_line()) {
        ...
}
```

Use array assignment to a local list to name your formal arguments:

```
sub maybeset {
        local($key, $value) = @_;
        $foo{$key} = $value unless $foo{$key};
}
```

This also has the effect of turning call-by-reference into call-by-value, since the assignment copies the values.

Subroutines may be called recursively. If a subroutine is called using the & form, the argument list is optional. If omitted, no @_ array is set up for the subroutine; the @_ array at the time of the call is visible to subroutine instead.

```
do foo(1,2,3);          # pass three arguments
&foo(1,2,3);            # the same

do foo();               # pass a null list
&foo();                 # the same
&foo;                   # pass no arguments—more efficient
```

Passing By Reference

Sometimes you don't want to pass the value of an array to a subroutine but rather the name of it, so that the subroutine can modify the global copy of it rather than working with a local copy. In perl you can refer to all the objects of a particular name by prefixing the name with a star: *foo. When evaluated, it produces a scalar value that represents all the objects of that name, including any filehandle, format or subroutine. When assigned to within a local() operation, it causes the name mentioned to refer to whatever * value was assigned to it. Example:

```
sub doubleary {
   local(*someary) = @_;
```

```
    foreach $elem (@someary) {
        $elem *= 2;
    }
}
do doubleary(*foo);
do doubleary(*bar);
```

Assignment to *name is currently recommended only inside a local(). You can actually assign to *name anywhere, but the previous referent of *name may be stranded forever. This may or may not bother you.

Note that scalars are already passed by reference, so you can modify scalar arguments without using this mechanism by referring explicitly to the $_[nnn] in question. You can modify all the elements of an array by passing all the elements as scalars, but you have to use the * mechanism to push, pop or change the size of an array. The * mechanism will probably be more efficient in any case.

Since a *name value contains unprintable binary data, if it is used as an argument in a print, or as a %s argument in a printf or sprintf, it then has the value '*name', just so it prints out pretty.

Even if you don't want to modify an array, this mechanism is useful for passing multiple arrays in a single LIST, since normally the LIST mechanism will merge all the array values so that you can't extract out the individual arrays.

Regular Expressions

The patterns used in pattern matching are regular expressions such as those supplied in the Version 8 regexp routines. (In fact, the routines are derived from Henry Spencer's freely redistributable reimplementation of the V8 routines.) In addition, \w matches an alphanumeric character (including "_") and \W a nonalphanumeric. Word boundaries may be matched by \b, and non-boundaries by \B. A whitespace character is matched by \s, non-whitespace by \S. A numeric character is matched by \d, non-numeric by \D. You may use \w, \s and \d within character classes. Also, \n, \r, \f, \t and \NNN have their normal interpretations. Within character classes \b represents backspace rather than a word boundary. Alternatives may be separated by |. The bracketing construct (...) may also be used, in which case \<digit> matches the digit'th substring. (Outside of the pattern, always use $ instead of \ in front of the digit. The scope of $<digit> (and $`, $& and $´) extends to the end of the enclosing BLOCK or eval string, or to the next pattern match with subexpressions. The \<digit> notation sometimes works outside the current pattern, but should not be relied upon.) You may have as many parentheses as you wish. If you have more than 9 substrings, the variables $10, $11, ... refer to the corresponding substring. Within the pattern, \10, \11, etc. refer back to substrings if there have been at least that many left parens before the backreference. Otherwise (for backward compatibilty) \10 is the same as \010, a backspace, and \11 the same as \011, a tab. And so on. (\1 through \9 are always backreferences.)

$+ returns whatever the last bracket match matched. $& returns the entire matched string. ($0 used to return the same thing, but not any more.) $` returns everything before the matched string. $´ returns everything after the matched string. Examples:

```
s/^([^ ]*) *([^ ]*)/$2 $1/;    # swap first two words

if (/Time: (..):(..):(..)/) {
        $hours = $1;
        $minutes = $2;
        $seconds = $3;
}
```

By default, the ^ character is only guaranteed to match at the beginning of the string, the $ character only at the end (or before the newline at the end) and *perl* does certain optimizations with the assumption that the string contains only one line. The behavior of ^ and $ on embedded newlines will be inconsistent. You may, however, wish to treat a string as a multi-line buffer, such that the ^ will match after any newline within the string, and $ will match before any newline. At the cost of a little more overhead, you can do this by setting the variable $* to 1. Setting it back to 0 makes *perl* revert to its old behavior.

To facilitate multi-line substitutions, the . character never matches a newline (even when $* is 0). In particular, the following leaves a newline on the $_ string:

```
$_ = <STDIN>;
s/.*(some_string).*/$1/;
```

If the newline is unwanted, try one of

```
s/.*(some_string).*\n/$1/;
s/.*(some_string)[^\000]*/$1/;
s/.*(some_string)(.|\n)*/$1/;
chop; s/.*(some_string).*/$1/;
/(some_string)/ && ($_ = $1);
```

Any item of a regular expression may be followed with digits in curly brackets of the form {n,m}, where n gives the minimum number of times to match the item and m gives the maximum. The form {n} is equivalent to {n,n} and matches exactly n times. The form {n,} matches n or more times. (If a curly bracket occurs in any other context, it is treated as a regular character.) The * modifier is equivalent to {0,}, the + modifier to {1,} and the ? modifier to {0,1}. There is no limit to the size of n or m, but large numbers will chew up more memory.

You will note that all backslashed metacharacters in *perl* are alphanumeric, such as \b, \w, \n. Unlike some other regular expression languages, there are no backslashed symbols that aren't alphanumeric. So anything that looks like \\, \(, \), \<, \>, \{, or \} is always interpreted as a literal character, not a metacharacter. This makes it simple to quote a string that you want to use for a pattern but that you are afraid might contain metacharacters. Simply quote all the non-alphanumeric characters:

```
$pattern =~ s/(\W)/\\$1/g;
```

Formats

Output record formats for use with the *write* operator may declared as follows:

```
format NAME =
FORMLIST
```

If name is omitted, format "STDOUT" is defined. FORMLIST consists of a sequence of lines, each of which may be of one of three types:

1. A comment.
2. A "picture" line giving the format for one output line.
3. An argument line supplying values to plug into a picture line.

Picture lines are printed exactly as they look, except for certain fields that substitute values into the line. Each picture field starts with either @ or ^. The @ field (not to be confused with the array marker @) is the normal case; ^ fields are used to do rudimentary multi-line text block filling. The length of the field is supplied by padding out the field with multiple <, >, or | characters to specify, respectively, left justification, right justification, or centering. As an alternate form of right justification, you may also use # characters (with an optional .) to specify a numeric field. (Use of ^ instead of @ causes the field to be blanked if undefined.) If any of the values supplied for these fields contains a newline, only the text up to the newline is printed. The special field @* can be used for printing multi-line values. It should appear by itself on a line.

The values are specified on the following line, in the same order as the picture fields. The values should be separated by commas.

Picture fields that begin with ^ rather than @ are treated specially. The value supplied must be a scalar variable name which contains a text string. *Perl* puts as much text as it can into the field, and then chops off the front of the string so that the next time the variable is referenced, more of the text can be printed. Normally you would use a sequence of fields in a vertical stack to print out a block of text. If you like, you can end the final field with ..., which will appear in the output if the text was too long to appear in its entirety. You can change which characters are legal to break on by changing the variable $: to a list of the desired characters.

Since use of ^ fields can produce variable length records if the text to be formatted is short, you can suppress blank lines by putting the tilde (~) character anywhere in the line. (Normally you should put it in the front if possible, for visibility.) The tilde will be translated to a space upon output. If you put a second tilde contiguous to the first, the line will be repeated until all the fields on the line are exhausted. (If you use a field of the @ variety, the expression you supply had better not give the same value every time forever!)

Examples:

```
# a report on the /etc/passwd file
format STDOUT_TOP =
                              Passwd File
Name                     Login    Office    Uid   Gid Home
------------------------------------------------------------------------------
.
format STDOUT =
@<<<<<<<<<<<<<<<<<<<<< @|||||| @<<<<<<@>>>> @>>>> @<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$name,                 $login,  $office,$uid,$gid, $home
.
```

```
# a report from a bug report form
format STDOUT_TOP =
                              Bug Reports
@<<<<<<<<<<<<<<<<<<<<<<<<     @|||          @>>>>>>>>>>>>>>>>>>>>>>>>
$system,                      $%,          $date
------------------------------------------------------------------------
.
format STDOUT =
Subject: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
         $subject
Index: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<  ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
       $index,                           $description
Priority: @<<<<<<<<<< Date: @<<<<<<<<   ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
          $priority,         $date,      $description
From: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<  ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
      $from,                            $description
Assigned to: @<<<<<<<<<<<<<<<<<<<<<<<<  ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
             $programmer,                $description
~                                        ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                         $description
~                                        ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                         $description
~                                        ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                         $description
~                                        ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                         $description
~                                        ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<...
                                         $description
.
```

It is possible to intermix prints with writes on the same output channel, but you'll have to handle $- (lines left on the page) yourself.

If you are printing lots of fields that are usually blank, you should consider using the reset operator between records. Not only is it more efficient, but it can prevent the bug of adding another field and forgetting to zero it.

Interprocess Communication

The IPC facilities of perl are built on the Berkeley socket mechanism. If you don't have sockets, you can ignore this section. The calls have the same names as the corresponding system calls, but the arguments tend to differ, for two reasons. First, perl file handles work differently than C file descriptors. Second, perl already knows the length of its strings, so you don't need to pass that information. Here is a sample client (untested):

```
($them,$port) = @ARGV;
$port = 2345 unless $port;
$them = 'localhost' unless $them;

$SIG{'INT'} = 'dokill';
sub dokill { kill 9,$child if $child; }

require 'sys/socket.ph';

$sockaddr = 'S n a4 x8';
```

```
chop($hostname = 'hostname');

($name, $aliases, $proto) = getprotobyname('tcp');
($name, $aliases, $port) = getservbyname($port, 'tcp')
        unless $port =~ /^\d+$/;
($name, $aliases, $type, $len, $thisaddr) = gethostbyname($hostname);
($name, $aliases, $type, $len, $thataddr) = gethostbyname($them);

$this = pack($sockaddr, &AF_INET, 0, $thisaddr);
$that = pack($sockaddr, &AF_INET, $port, $thataddr);

socket(S, &PF_INET, &SOCK_STREAM, $proto) || die "socket: $!";
bind(S, $this) || die "bind: $!";
connect(S, $that) || die "connect: $!";

select(S); $| = 1; select(stdout);

if ($child = fork) {
        while (<>) {
                print S;
        }
        sleep 3;
        do dokill();
}
else {
        while (<S>) {
                print;
        }
}
```

And here's a server:

```
($port) = @ARGV;
$port = 2345 unless $port;

require 'sys/socket.ph';

$sockaddr = 'S n a4 x8';

($name, $aliases, $proto) = getprotobyname('tcp');
($name, $aliases, $port) = getservbyname($port, 'tcp')
        unless $port =~ /^\d+$/;

$this = pack($sockaddr, &AF_INET, $port, "\0\0\0\0");

select(NS); $| = 1; select(stdout);

socket(S, &PF_INET, &SOCK_STREAM, $proto) || die "socket: $!";
bind(S, $this) || die "bind: $!";
listen(S, 5) || die "connect: $!";

select(S); $| = 1; select(stdout);
```

```
for (;;) {
        print "Listening again\n";
        ($addr = accept(NS,S)) || die $!;
        print "accept ok\n";

        ($af,$port,$inetaddr) = unpack($sockaddr,$addr);
        @inetaddr = unpack('C4',$inetaddr);
        print "$af $port @inetaddr\n";

        while (<NS>) {
                print;
                print NS;
        }
}
```

Predefined Names

The following names have special meaning to *perl*. I could have used alphabetic symbols for some of these, but I didn't want to take the chance that someone would say reset "a–zA–Z" and wipe them all out. You'll just have to suffer along with these silly symbols. Most of them have reasonable mnemonics, or analogues in one of the shells.

$_ The default input and pattern-searching space. The following pairs are equivalent:

```
while (<>) {...          # only equivalent in while!
while ($_ = <>) {...

/^Subject:/
$_ =~ /^Subject:/

y/a-z/A-Z/
$_ =~ y/a-z/A-Z/

chop
chop($_)
```

(Mnemonic: underline is understood in certain operations.)

$. The current input line number of the last filehandle that was read. Readonly. Remember that only an explicit close on the filehandle resets the line number. Since <> never does an explicit close, line numbers increase across ARGV files (but see examples under eof). (Mnemonic: many programs use . to mean the current line number.)

$/ The input record separator, newline by default. Works like *awk*'s RS variable, including treating blank lines as delimiters if set to the null string. You may set it to a multicharacter string to match a multi-character delimiter. (Mnemonic: / is used to delimit line boundaries when quoting poetry.)

$, The output field separator for the print operator. Ordinarily the print operator simply prints out the comma separated fields you specify. In order to get behavior more like *awk*, set this variable as you would set *awk*'s OFS variable to specify what is printed between fields. (Mnemonic: what is printed when there is a , in your print statement.)

$" This is like $, except that it applies to array values interpolated into a double-quoted string (or similar interpreted string). Default is a space. (Mnemonic: obvious, I think.)

$\ The output record separator for the print operator. Ordinarily the print operator simply prints out the comma separated fields you specify, with no trailing newline or record separator assumed. In order to get behavior more like *awk*, set this variable as you would set *awk*'s ORS variable to specify what is printed at the end of the print. (Mnemonic: you set $\ instead of adding \n at the end of the print. Also, it's just like /, but it's what you get "back" from *perl*.)

$# The output format for printed numbers. This variable is a half-hearted attempt to emulate *awk*'s OFMT variable. There are times, however, when *awk* and *perl* have differing notions of what is in fact numeric. Also, the initial value is %.20g rather than %.6g, so you need to set $# explicitly to get *awk*'s value. (Mnemonic: # is the number sign.)

$% The current page number of the currently selected output channel. (Mnemonic: % is page number in nroff.)

$= The current page length (printable lines) of the currently selected output channel. Default is 60. (Mnemonic: = has horizontal lines.)

$- The number of lines left on the page of the currently selected output channel. (Mnemonic: lines_on_page - lines_printed.)

$~ The name of the current report format for the currently selected output channel. Default is name of the filehandle. (Mnemonic: brother to $^.)

$^ The name of the current top-of-page format for the currently selected output channel. Default is name of the filehandle with "_TOP" appended. (Mnemonic: points to top of page.)

$| If set to nonzero, forces a flush after every write or print on the currently selected output channel. Default is 0. Note that *STDOUT* will typically be line buffered if output is to the terminal and block buffered otherwise. Setting this variable is useful primarily when you are outputting to a pipe, such as when you are running a *perl* script under rsh and want to see the output as it's happening. (Mnemonic: when you want your pipes to be piping hot.)

$$ The process number of the *perl* running this script. (Mnemonic: same as shells.)

$? The status returned by the last pipe close, backtick (``) command or *system* operator. Note that this is the status word returned by the wait() system call, so the exit value of the subprocess is actually ($? >> 8). $? & 255 gives which signal, if any, the process died from, and whether there was a core dump. (Mnemonic: similar to sh and ksh.)

$& The string matched by the last pattern match (not counting any matches hidden within a BLOCK or eval enclosed by the current BLOCK). (Mnemonic: like & in some editors.)

$` The string preceding whatever was matched by the last pattern match (not counting any matches hidden within a BLOCK or eval enclosed by the current BLOCK). (Mnemonic: ` often precedes a quoted string.)

$´ The string following whatever was matched by the last pattern match (not counting any matches hidden within a BLOCK or eval enclosed by the current BLOCK). (Mnemonic: ´ often follows a quoted string.) Example:

```
$_ = 'abcdefghi';
/def/;
print "$`:$&:$'\n";      # prints abc:def:ghi
```

$+ The last bracket matched by the last search pattern. This is useful if you don't know which of a set of alternative patterns matched. For example:

```
/Version: (.*)|Revision: (.*)/ && ($rev = $+);
```

(Mnemonic: be positive and forward looking.)

$* Set to 1 to do multiline matching within a string, 0 to tell *perl* that it can assume that strings contain a single line, for the purpose of optimizing pattern matches. Pattern matches on strings containing multiple newlines can produce confusing results when $* is 0. Default is 0. (Mnemonic: * matches multiple things.) Note that this variable only influences the interpretation of ^ and $. A literal newline can be searched for even when $* == 0.

$0 Contains the name of the file containing the *perl* script being executed. Assigning to $0 modifies the argument area that the ps(1) program sees. (Mnemonic: same as sh and ksh.)

$<digit>
Contains the subpattern from the corresponding set of parentheses in the last pattern matched, not counting patterns matched in nested blocks that have been exited already. (Mnemonic: like \digit.)

$[The index of the first element in an array, and of the first character in a substring. Default is 0, but you could set it to 1 to make *perl* behave more like *awk* (or Fortran) when subscripting and when evaluating the index() and substr() functions. (Mnemonic: [begins subscripts.)

$] The string printed out when you say "perl -v". It can be used to determine at the beginning of a script whether the perl interpreter executing the script is in the right range of versions. If used in a numeric context, returns the version + patchlevel / 1000. Example:

```
    # see if getc is available
($version,$patchlevel) =
            $] =~ /(\d+\.\d+).*\nPatch level: (\d+)/;
print STDERR "(No filename completion available.)\n"
            if $version * 1000 + $patchlevel < 2016;
```

or, used numerically,

```
warn "No checksumming!\n" if $] < 3.019;
```

(Mnemonic: Is this version of perl in the right bracket?)

$; The subscript separator for multi-dimensional array emulation. If you refer to an associative array element as

```
$foo{$a,$b,$c}
```

it really means

```
$foo{join($;, $a, $b, $c)}
```

But don't put

```
@foo{$a,$b,$c}          # a slice--note the @
```

which means

```
($foo{$a},$foo{$b},$foo{$c})
```

Default is "\034", the same as SUBSEP in *awk*. Note that if your keys contain binary data there might not be any safe value for $;. (Mnemonic: comma (the syntactic subscript separator) is a semi-semicolon. Yeah, I know, it's pretty lame, but $, is already taken for something more important.)

$! If used in a numeric context, yields the current value of errno, with all the usual caveats. (This means that you shouldn't depend on the value of $! to be anything in particular unless you've gotten a specific error return indicating a system error.) If used in a string context, yields the corresponding system error string. You can assign to $! in order to set errno if, for instance, you want $! to return the string for error n, or you want to set the exit value for the die operator. (Mnemonic: What just went bang?)

$@ The perl syntax error message from the last eval command. If null, the last eval parsed and executed correctly (although the operations you invoked may have failed in the normal fashion). (Mnemonic: Where was the syntax error "at"?)

$< The real uid of this process. (Mnemonic: it's the uid you came FROM, if you're running setuid.)

$> The effective uid of this process. Example:

```
$< = $>;       # set real uid to the effective uid
($<,$>) = ($>,$<);  # swap real and effective uid
```

(Mnemonic: it's the uid you went TO, if you're running setuid.) Note: $< and $> can only be swapped on machines supporting setreuid().

$(The real gid of this process. If you are on a machine that supports membership in multiple groups simultaneously, gives a space separated list of groups you are in. The first number is the one returned by getgid(), and the subsequent ones by getgroups(), one of which may be the same as the first number. (Mnemonic: parentheses are used to GROUP things. The real gid is the group you LEFT, if you're running setgid.)

$) The effective gid of this process. If you are on a machine that supports membership in multiple groups simultaneously, gives a space separated list of groups you are in. The first number is the one returned by getegid(), and the subsequent ones by getgroups(), one of which may be the same as the first number. (Mnemonic: parentheses are used to GROUP things. The effective gid is the group that's RIGHT for you, if you're running setgid.)

Note: $<, $>, $(and $) can only be set on machines that support the corresponding set[re][ug]id() routine. $(and $) can only be swapped on machines supporting setregid().

$: The current set of characters after which a string may be broken to fill continuation fields (starting with ^) in a format. Default is " \n-", to break on whitespace or hyphens. (Mnemonic: a "colon" in poetry is a part of a line.)

$^D The current value of the debugging flags. (Mnemonic: value of –**D** switch.)

$^F The maximum system file descriptor, ordinarily 2. System file descriptors are passed to subprocesses, while higher file descriptors are not. During an open, system file descriptors are preserved even if the open fails. Ordinary file descriptors are closed before the open is attempted.

$^I The current value of the inplace-edit extension. Use undef to disable inplace editing. (Mnemonic: value of –i switch.)

$^P The internal flag that the debugger clears so that it doesn't debug itself. You could conceivable disable debugging yourself by clearing it.

$^T The time at which the script began running, in seconds since the epoch. The values returned by the –M , –A and –C filetests are based on this value.

$^W The current value of the warning switch. (Mnemonic: related to the –w switch.)

$^X The name that Perl itself was executed as, from argv[0].

$ARGV contains the name of the current file when reading from <>.

@ARGV
The array ARGV contains the command line arguments intended for the script. Note that $#ARGV is the generally number of arguments minus one, since $ARGV[0] is the first argument, NOT the command name. See $0 for the command name.

@INC The array INC contains the list of places to look for *perl* scripts to be evaluated by the "do EXPR" command or the "require" command. It initially consists of the arguments to any –I command line switches, followed by the default *perl* library, probably "/usr/local/lib/perl", followed by ".", to represent the current directory.

%INC The associative array INC contains entries for each filename that has been included via "do" or "require". The key is the filename you specified, and the value is the location of the file actually found. The "require" command uses this array to determine whether a given file has already been included.

$ENV{expr}
The associative array ENV contains your current environment. Setting a value in ENV changes the environment for child processes.

$SIG{expr}
The associative array SIG is used to set signal handlers for various signals. Example:

```
sub handler {    # 1st argument is signal name
        local($sig) = @_;
        print "Caught a SIG$sig--shutting down\n";
        close(LOG);
        exit(0);
}

$SIG{'INT'} = 'handler';
$SIG{'QUIT'} = 'handler';
...
$SIG{'INT'} = 'DEFAULT';   # restore default action
$SIG{'QUIT'} = 'IGNORE';   # ignore SIGQUIT
```

The SIG array only contains values for the signals actually set within the perl script.

Packages

Perl provides a mechanism for alternate namespaces to protect packages from stomping on each others variables. By default, a perl script starts compiling into the package known as "main". By use of the *package* declaration, you can switch namespaces. The scope of the package declaration is from the declaration itself to the end of the enclosing block (the same scope as the local() operator). Typically it would be the first declaration in a file to be included by the "require" operator. You can switch into a package in more than one place; it merely influences which symbol table is used by the compiler for the rest of that block. You can refer to variables and

filehandles in other packages by prefixing the identifier with the package name and a single quote. If the package name is null, the "main" package as assumed.

Only identifiers starting with letters are stored in the packages symbol table. All other symbols are kept in package "main". In addition, the identifiers STDIN, STDOUT, STDERR, ARGV, ARGVOUT, ENV, INC and SIG are forced to be in package "main", even when used for other purposes than their built-in one. Note also that, if you have a package called "m", "s" or "y", the you can't use the qualified form of an identifier since it will be interpreted instead as a pattern match, a substitution or a translation.

Eval'ed strings are compiled in the package in which the eval was compiled in. (Assignments to $SIG{}, however, assume the signal handler specified is in the main package. Qualify the signal handler name if you wish to have a signal handler in a package.) For an example, examine perldb.pl in the perl library. It initially switches to the DB package so that the debugger doesn't interfere with variables in the script you are trying to debug. At various points, however, it temporarily switches back to the main package to evaluate various expressions in the context of the main package.

The symbol table for a package happens to be stored in the associative array of that name prepended with an underscore. The value in each entry of the associative array is what you are referring to when you use the *name notation. In fact, the following have the same effect (in package main, anyway), though the first is more efficient because it does the symbol table lookups at compile time:

```
local(*foo) = *bar;
local($_main{'foo'}) = $_main{'bar'};
```

You can use this to print out all the variables in a package, for instance. Here is dumpvar.pl from the perl library:

```
package dumpvar;

sub main'dumpvar {
    ($package) = @_;
    local(*stab) = eval("*_$package");
    while (($key,$val) = each(%stab)) {
        {
            local(*entry) = $val;
            if (defined $entry) {
                print "\$$key = '$entry'\n";
            }
            if (defined @entry) {
                print "\@$key = (\n";
                foreach $num ($[ .. $#entry) {
                    print "  $num\t'",$entry[$num],"'\n";
                }
                print ")\n";
            }
```

```
            if ($key ne "_$package" && defined %entry) {
                print "\%$key = (\n";
                foreach $key (sort keys(%entry)) {
                    print "  $key\t'",$entry{$key},"'\n";
                }
                print ")\n";
            }
        }
    }
}
```

Note that, even though the subroutine is compiled in package dumpvar, the name of the subroutine is qualified so that its name is inserted into package "main".

Style

Each programmer will, of course, have his or her own preferences in regards to formatting, but there are some general guidelines that will make your programs easier to read.

1. Just because you CAN do something a particular way doesn't mean that you SHOULD do it that way. *Perl* is designed to give you several ways to do anything, so consider picking the most readable one. For instance

   ```
   open(FOO,$foo) || die "Can't open $foo: $!";
   ```

   is better than

   ```
   die "Can't open $foo: $!" unless open(FOO,$foo);
   ```

   because the second way hides the main point of the statement in a modifier. On the other hand

   ```
   print "Starting analysis\n" if $verbose;
   ```

   is better than

   ```
   $verbose && print "Starting analysis\n";
   ```

   since the main point isn't whether the user typed -v or not.

   Similarly, just because an operator lets you assume default arguments doesn't mean that you have to make use of the defaults. The defaults are there for lazy systems programmers writing one-shot programs. If you want your program to be readable, consider supplying the argument.

   Along the same lines, just because you *can* omit parentheses in many places doesn't mean that you ought to:

   ```
   return print reverse sort num values array;
   return print(reverse(sort num (values(%array))));
   ```

   When in doubt, parenthesize. At the very least it will let some poor schmuck bounce on the % key in vi.

   Even if you aren't in doubt, consider the mental welfare of the person who has to maintain the code after you, and who will probably put parens in the wrong place.

2. Don't go through silly contortions to exit a loop at the top or the bottom, when *perl* provides the "last" operator so you can exit in the middle. Just outdent it a little to make it more visible:

```
line:
    for (;;) {
        statements;
    last line if $foo;
        next line if /^#/;
        statements;
    }
```

3. Don't be afraid to use loop labels—they're there to enhance readability as well as to allow multi-level loop breaks. See last example.
4. For portability, when using features that may not be implemented on every machine, test the construct in an eval to see if it fails. If you know what version or patchlevel a particular feature was implemented, you can test $] to see if it will be there.
5. Choose mnemonic identifiers.
6. Be consistent.

Debugging

If you invoke *perl* with a **-d** switch, your script will be run under a debugging monitor. It will halt before the first executable statement and ask you for a command, such as:

h	Prints out a help message.
T	Stack trace.
s	Single step. Executes until it reaches the beginning of another statement.
n	Next. Executes over subroutine calls, until it reaches the beginning of the next statement.
f	Finish. Executes statements until it has finished the current subroutine.
c	Continue. Executes until the next breakpoint is reached.
c line	Continue to the specified line. Inserts a one-time-only breakpoint at the specified line.
<CR>	Repeat last n or s.
l min+incr	List incr+1 lines starting at min. If min is omitted, starts where last listing left off. If incr is omitted, previous value of incr is used.
l min-max	List lines in the indicated range.
l line	List just the indicated line.
l	List next window.
-	List previous window.
w line	List window around line.
l subname	List subroutine. If it's a long subroutine it just lists the beginning. Use "l" to list more.

/pattern/ Regular expression search forward for pattern; the final / is optional.

?pattern? Regular expression search backward for pattern; the final ? is optional.

L List lines that have breakpoints or actions.

S Lists the names of all subroutines.

t Toggle trace mode on or off.

b line condition
Set a breakpoint. If line is omitted, sets a breakpoint on the line that is about to be executed. If a condition is specified, it is evaluated each time the statement is reached and a breakpoint is taken only if the condition is true. Breakpoints may only be set on lines that begin an executable statement.

b subname condition
Set breakpoint at first executable line of subroutine.

d line Delete breakpoint. If line is omitted, deletes the breakpoint on the line that is about to be executed.

D Delete all breakpoints.

a line command
Set an action for line. A multi-line command may be entered by backslashing the newlines.

A Delete all line actions.

< command Set an action to happen before every debugger prompt. A multi-line command may be entered by backslashing the newlines.

> command Set an action to happen after the prompt when you've just given a command to return to executing the script. A multi-line command may be entered by backslashing the newlines.

V package List all variables in package. Default is main package.

! number Redo a debugging command. If number is omitted, redoes the previous command.

! -number Redo the command that was that many commands ago.

H -number Display last n commands. Only commands longer than one character are listed. If number is omitted, lists them all.

q or ^D Quit.

command Execute command as a perl statement. A missing semicolon will be supplied.

p expr Same as "print DB'OUT expr". The DB'OUT filehandle is opened to /dev/tty, regardless of where STDOUT may be redirected to.

If you want to modify the debugger, copy perldb.pl from the perl library to your current directory and modify it as necessary. (You'll also have to put -I. on your command line.) You can do some customization by setting up a .perldb file which contains initialization code. For instance, you could make aliases like these:

```
$DB'alias{'len'} = 's/^len(.*)/p length($1)/';
$DB'alias{'stop'} = 's/^stop (at|in)/b/';
$DB'alias{'.'} =
  's/^\./p "\$DB\'sub(\$DB\'line):\t",\$DB\'line[\$DB\'line]/';
```

Setuid Scripts

Perl is designed to make it easy to write secure setuid and setgid scripts. Unlike shells, which are based on multiple substitution passes on each line of the script, *perl* uses a more conventional evaluation scheme with fewer hidden "gotchas". Additionally, since the language has more built-in functionality, it has to rely less upon external (and possibly untrustworthy) programs to accomplish its purposes.

In an unpatched 4.2 or 4.3bsd kernel, setuid scripts are intrinsically insecure, but this kernel feature can be disabled. If it is, *perl* can emulate the setuid and setgid mechanism when it notices the otherwise useless setuid/gid bits on perl scripts. If the kernel feature isn't disabled, *perl* will complain loudly that your setuid script is insecure. You'll need to either disable the kernel setuid script feature, or put a C wrapper around the script.

When perl is executing a setuid script, it takes special precautions to prevent you from falling into any obvious traps. (In some ways, a perl script is more secure than the corresponding C program.) Any command line argument, environment variable, or input is marked as "tainted", and may not be used, directly or indirectly, in any command that invokes a subshell, or in any command that modifies files, directories or processes. Any variable that is set within an expression that has previously referenced a tainted value also becomes tainted (even if it is logically impossible for the tainted value to influence the variable). For example:

```
$foo = shift;                   # $foo is tainted
$bar = $foo,'bar';              # $bar is also tainted
$xxx = <>;                      # Tainted
$path = $ENV{'PATH'};           # Tainted, but see below
$abc = 'abc';                   # Not tainted

system "echo $foo";             # Insecure
system "/bin/echo", $foo;       # Secure (doesn't use sh)
system "echo $bar";             # Insecure
system "echo $abc";             # Insecure until PATH set

$ENV{'PATH'} = '/bin:/usr/bin';
$ENV{'IFS'} = '' if $ENV{'IFS'} ne '';

$path = $ENV{'PATH'};           # Not tainted
system "echo $abc";             # Is secure now!

open(FOO,"$foo");               # OK
open(FOO,">$foo");              # Not OK

open(FOO,"echo $foo|");         # Not OK, but...
open(FOO,"-|") || exec 'echo', $foo;    # OK

$zzz = `echo $foo`;             # Insecure, zzz tainted

unlink $abc,$foo;               # Insecure
umask $foo;                     # Insecure

exec "echo $foo";               # Insecure
exec "echo", $foo;              # Secure (doesn't use sh)
exec "sh", '-c', $foo;    # Considered secure, alas
```

The taintedness is associated with each scalar value, so some elements of an array can be tainted,

and others not.

If you try to do something insecure, you will get a fatal error saying something like "Insecure dependency" or "Insecure PATH". Note that you can still write an insecure system call or exec, but only by explicitly doing something like the last example above. You can also bypass the tainting mechanism by referencing subpatterns—*perl* presumes that if you reference a substring using $1, $2, etc, you knew what you were doing when you wrote the pattern:

```
$ARGV[0] =~ /^-P(\w+)$/;
$printer = $1;          # Not tainted
```

This is fairly secure since \w+ doesn't match shell metacharacters. Use of .+ would have been insecure, but *perl* doesn't check for that, so you must be careful with your patterns. This is the ONLY mechanism for untainting user supplied filenames if you want to do file operations on them (unless you make $> equal to $<).

It's also possible to get into trouble with other operations that don't care whether they use tainted values. Make judicious use of the file tests in dealing with any user-supplied filenames. When possible, do opens and such after setting $> = $<. *Perl* doesn't prevent you from opening tainted filenames for reading, so be careful what you print out. The tainting mechanism is intended to prevent stupid mistakes, not to remove the need for thought.

ENVIRONMENT

Perl uses PATH in executing subprocesses, and in finding the script if -S is used. HOME or LOGDIR are used if chdir has no argument.

Apart from these, *perl* uses no environment variables, except to make them available to the script being executed, and to child processes. However, scripts running setuid would do well to execute the following lines before doing anything else, just to keep people honest:

```
$ENV{'PATH'} = '/bin:/usr/bin';    # or whatever you need
$ENV{'SHELL'} = '/bin/sh' if $ENV{'SHELL'} ne '';
$ENV{'IFS'} = '' if $ENV{'IFS'} ne '';
```

AUTHOR

Larry Wall <lwall@jpl-devvax.Jpl.Nasa.Gov>
MS-DOS port by Diomidis Spinellis <dds@cc.ic.ac.uk>

FILES

/tmp/perl-eXXXXXX temporary file for -e commands.

SEE ALSO

a2p awk to perl translator
s2p sed to perl translator

DIAGNOSTICS

Compilation errors will tell you the line number of the error, with an indication of the next token or token type that was to be examined. (In the case of a script passed to *perl* via -e switches, each -e is counted as one line.)

Setuid scripts have additional constraints that can produce error messages such as "Insecure dependency". See the section on setuid scripts.

TRAPS

Accustomed *awk* users should take special note of the following:

* Semicolons are required after all simple statements in *perl*. Newline is not a statement delimiter.

* Curly brackets are required on ifs and whiles.
* Variables begin with $ or @ in *perl*.
* Arrays index from 0 unless you set $[. Likewise string positions in substr() and index().
* You have to decide whether your array has numeric or string indices.
* Associative array values do not spring into existence upon mere reference.
* You have to decide whether you want to use string or numeric comparisons.
* Reading an input line does not split it for you. You get to split it yourself to an array. And the *split* operator has different arguments.
* The current input line is normally in $_, not $0. It generally does not have the newline stripped. ($0 is the name of the program executed.)
* $<digit> does not refer to fields—it refers to substrings matched by the last match pattern.
* The *print* statement does not add field and record separators unless you set $, and $\.
* You must open your files before you print to them.
* The range operator is "..", not comma. (The comma operator works as in C.)
* The match operator is "=~", not "~". ("~" is the one's complement operator, as in C.)
* The exponentiation operator is "**", not "^". ("^" is the XOR operator, as in C.)
* The concatenation operator is ".", not the null string. (Using the null string would render "/pat/ /pat/" unparsable, since the third slash would be interpreted as a division operator—the tokener is in fact slightly context sensitive for operators like /, ?, and <. And in fact, . itself can be the beginning of a number.)
* *Next*, *exit* and *continue* work differently.
* The following variables work differently

Awk	Perl
ARGC	$#ARGV
ARGV[0]	$0
FILENAME	$ARGV
FNR	$. – something
FS	(whatever you like)
NF	$#Fld, or some such
NR	$.
OFMT	$#
OFS	$,
ORS	$\
RLENGTH	length($&)
RS	$/
RSTART	length($`)
SUBSEP	$;

* When in doubt, run the *awk* construct through a2p and see what it gives you.

Cerebral C programmers should take note of the following:

* Curly brackets are required on ifs and whiles.
* You should use "elsif" rather than "else if"
* *Break* and *continue* become *last* and *next*, respectively.

* There's no switch statement.
* Variables begin with $ or @ in *perl*.
* Printf does not implement *.
* Comments begin with #, not /*.
* You can't take the address of anything.
* ARGV must be capitalized.
* The "system" calls link, unlink, rename, etc. return nonzero for success, not 0.
* Signal handlers deal with signal names, not numbers.

Seasoned *sed* programmers should take note of the following:

* Backreferences in substitutions use $ rather than \.
* The pattern matching metacharacters (,), and | do not have backslashes in front.
* The range operator is .. rather than comma.

Sharp shell programmers should take note of the following:

* The backtick operator does variable interpretation without regard to the presence of single quotes in the command.
* The backtick operator does no translation of the return value, unlike csh.
* Shells (especially csh) do several levels of substitution on each command line. *Perl* does substitution only in certain constructs such as double quotes, backticks, angle brackets and search patterns.
* Shells interpret scripts a little bit at a time. *Perl* compiles the whole program before executing it.
* The arguments are available via @ARGV, not $1, $2, etc.
* The environment is not automatically made available as variables.

ERRATA AND ADDENDA

The Perl book, *Programming Perl*, has the following omissions and goofs.

On page 5, the examples which read

```
eval "/usr/bin/perl
```

should read

```
eval "exec /usr/bin/perl
```

On page 195, the equivalent to the System V sum program only works for very small files. To do larger files, use

```
undef $/;
$checksum = unpack("%32C*",<>) % 32767;
```

The **–0** switch to set the initial value of $/ was added to Perl after the book went to press.

The **–l** switch now does automatic line ending processing.

The qx// construct is now a synonym for backticks.

$0 may now be assigned to set the argument displayed by *ps (1)*.

The new @###.## format was omitted accidentally from the description on formats.

It wasn't known at press time that s///ee caused multiple evaluations of the replacement expression. This is to be construed as a feature.

(LIST) x $count now does array replication.

There is now no limit on the number of parentheses in a regular expression.

In double-quote context, more escapes are supported: \e, \a, \x1b, \c[, \l, \L, \u, \U, \E. The latter five control up/lower case translation.

The $/ variable may now be set to a multi-character delimiter.

BUGS

Perl is at the mercy of your machine's definitions of various operations such as type casting, atof() and sprintf().

If your stdio requires an seek or eof between reads and writes on a particular stream, so does *perl*. (This doesn't apply to sysread() and syswrite().)

While none of the built-in data types have any arbitrary size limits (apart from memory size), there are still a few arbitrary limits: a given identifier may not be longer than 255 characters; sprintf is limited on many machines to 128 characters per field (unless the format specifier is exactly %s); and no component of your PATH may be longer than 255 if you use –S.

Perl actually stands for Pathologically Eclectic Rubbish Lister, but don't tell anyone I said that.

NAME

plot – graphics filters

SYNOPSIS

plot [**–T**terminal [raster]]

DESCRIPTION

These commands read plotting instructions (see *plot*(5)) from the standard input, and in general produce plotting instructions suitable for a particular *terminal* on the standard output.

If no *terminal* type is specified, the environment parameter $TERM (see *environ*(7)) is used. If the environment parameter $TERM is not set then the Tektronix 4014 storage scope is the default. Known *terminals* are:

4014 Tektronix 4014 storage scope.

450 DASI Hyterm 450 terminal (Diablo mechanism).

300 DASI 300 or GSI terminal (Diablo mechanism).

300S DASI 300S terminal (Diablo mechanism).

ver Versatec D1200A printer-plotter. 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 optional argument causes a previously scan-converted file *raster* to be sent to the plotter.

FILES

/usr/bin/tek
/usr/bin/t450
/usr/bin/t300
/usr/bin/t300s
/usr/bin/vplot
/usr/tmp/raster

SEE ALSO

plot(3X), plot(5)

BUGS

There is no lockout protection for */usr/tmp/raster*.

NAME

pr – print file

SYNOPSIS

pr [option] ... [file] ...

DESCRIPTION

Pr produces a printed listing of one or more *files*. The output is separated into pages headed by a date, the name of the file or a specified header, and the page number. If there are no file arguments, *pr* prints its standard input.

Options apply to all following files but may be reset between files:

–*n* Produce *n*-column output.

+*n* Begin printing with page *n*.

–h Take the next argument as a page header.

–w*n* For purposes of multi-column output, take the width of the page to be *n* characters instead of the default 72.

–f Use formfeeds instead of newlines to separate pages. A formfeed is assumed to use up two blank lines at the top of a page. (Thus this option does not affect the effective page length.)

–l*n* Take the length of the page to be *n* lines instead of the default 66.

–t Do not print the 5-line header or the 5-line trailer normally supplied for each page.

–s*c* Separate columns by the single character *c* instead of by the appropriate amount of white space. A missing *c* is taken to be a tab.

–m Print all *files* simultaneously, each in one column,

Inter-terminal messages via *write*(1) are forbidden during a *pr*.

FILES

/dev/tty? to suspend messages.

SEE ALSO

cat(1)

DIAGNOSTICS

There are no diagnostics when *pr* is printing on a terminal.

NAME

print – pr to the line printer

SYNOPSIS

print file ...

DESCRIPTION

Print pr's a copy of each named file on the line printer. It is a one line shell script:

```
lpr -p $*
```

Since all of the arguments given to **print** are passed along to **lpr** uninterpreted, **lpr** options may be given to **print**.

SEE ALSO

lpr(1), pr(1)

NAME

printenv – print out the environment

SYNOPSIS

printenv [name]

DESCRIPTION

Printenv prints out the values of the variables in the environment. If a *name* is specified, only its value is printed.

If a *name* is specified and it is not defined in the environment, *printenv* returns exit status 1, else it returns status 0.

SEE ALSO

sh(1), environ(7), csh(1)

NAME

prmail – print out mail in the post office

SYNOPSIS

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/spool/mail/* post office

SEE ALSO

biff(1), mail(1), from(1), binmail(1)

NAME

prx – filter for Printronix printers

SYNOPSIS

prx [**–l***n*] [**–h***n*] [**–v***n*] [**–t***n*] files

DESCRIPTION

Prx filters nroff output for Printronix printers. More specifically, it handles underlining and character translation for the Greek character set. Although the backspace character is interpreted normally (the Printronix uses it to produce double-size characters), the Printronix doesn't support overstriking, so only the last character at a location will be printed.

The options available include the following:

-l*n* Set the page size to *n* lines. The default is set to 56 (same as the Qume printer).

–h*n* Set horizontal margin to *n* columns. Default is 10.

–v*n* Set the position of the first line on the page to *n* lines from the top edge of the paper. Default is 5.

–t*n* Set tab size to *n* spaces. Default is 8 spaces.

SEE ALSO

lpr(1), nroff(1)

NAME

ps – process status

SYNOPSIS

ps [**acegklstuvwx#**]

DESCRIPTION

ps prints information about processes. Normally, only your processes are candidates to be printed by *ps*. Specifying **a** causes other users processes to be candidates to be printed; specifying **x** includes processes without control terminals in the candidate pool.

OPTIONS

The command-line options are:

a Asks for information about all processes with terminals (ordinarily only one's own processes are displayed).

c Prints the command name, as stored internally in the system for purposes of accounting, rather than the command arguments, which are kept in the process' address space. This is more reliable, if less informative, since the process is free to destroy the latter information.

e Asks for the environment to be printed as well as the arguments to the command.

g Asks for all processes. Without this option, *ps* only prints **interesting** processes. Processes are deemed to be uninteresting if they are process group leaders. This normally eliminates top-level command interpreters and processes waiting for users to login on free terminals.

k Causes the file *mm.core* to be used in place of */dev/mem* and *vmunix* to be used instead of */vmunix*. This is used for postmortem system debugging.

l Asks for a long listing, with fields F, UID, PPID, CP, PRI, NI, SZ, RSS, and WCHAN as described below.

s Adds the size SSIZ of the largest kernel stack of each process (for use by system maintainers) to the basic output format. All of the kernel stacks of a process (i.e. both the fault and system call stacks for each thread of the process) are considered in the determination of the largest stack.

t*x* Restricts output to processes whose controlling *tty* is *x*, which should be specified as printed by *ps* (e.g., *03* for *tty*03, *tco* for console, *t?* for processes with no *tty*, *t* for processes at the current *tty*, etc). This option must be the last one given.

u A user-oriented output is produced. This includes USER, %CPU, %MEM, SZ, and RSS, as described below.

v A version of the output containing virtual memory statistics is output. This includes SL, RE, PAGEIN, SZ, RSS, LIM, TSIZ, TRS, %CPU, and %MEM fields, as described below.

w Uses a wide output format (132 columns rather than 80); if repeated (e.g., **ww**) **ps** uses arbitrarily wide output. This information is used to decide how much of long commands to print.

x Asks about processes with no terminal.

A process number may be given (indicated here by **#**), in which case the output is restricted to that process. This option must also be last.

OTHER ARGUMENTS

A second argument is taken to be the file containing the system's namelist. Otherwise, */vmunix* is used. A third argument tells *ps* where to look for *core* if the **k** option is given, instead of */vmcore*. If a fourth argument is given, it is taken to be the name of a swap file, instead of the default */dev/drum*.

COMMON OUTPUT FIELDS

All output formats include these fields for each process:

PID Process id of the process. Each process in the system has its own unique pid.

TT The tty device which is "associated" with the process. (Also known as the controlling terminal for the process.) If the tty listed is **?**, then the process is not associated with any tty device. For example, daemon processes such as */etc/cron*

normally don't have a controlling tty. If the tty listed is **??**, then *ps* could not determine the controlling tty.

TIME The amount of CPU time which has been used by the process, including both user and system time.

STAT The state of the process. See paragraphs below for information on interpreting this field.

COMMAND The name of the command. Note that if for some reason *ps* can't access the correct pages of a process' stack, the command name for that process will be listed in parentheses.

PROCESS STATES

The state of the process is given by a sequence of five letters, e.g. **VRWNA**.

The first letter indicates whether a process is utilizing the vector processor. **V** indicates the vector processor is being utilized; a blank is shown if it is not.

The second letter indicates the runnability of the process:

R Indicates runnable processes.

T Indicates stopped processes.

P Indicates processes in page wait.

D Indicates processes in disk or other short term waits.

S Indicates active (sleeping for less than about 20 seconds) processes.

I Indicates idle (sleeping longer than about 20 seconds) processes.

Z Indicates zombied processes (trying to exit, but the process' parent is not waiting for it).

The third letter indicates whether a process is swapped out. It shows as **W** if it is, a blank if it is loaded (incore), or a **P** if it is partially swapped. A process which has specified a soft limit on memory requirements and which is exceeding that limit, shows **>**; such a process is necessarily not swapped.

The fourth letter indicates whether a process is running with altered CPU scheduling priority (*nice*). If the process priority is reduced, an **N** is shown; if the process priority has been artificially raised, a **<** is shown; processes running without special treatment have just a blank.

The fifth letter indicates any special treatment of the process for virtual memory replacement. The letters correspond to options to the *vadvise*(2) call. Currently the possibilities are: **A** (VA_ANOM) represents a *lisp*(1) in garbage collection, **S** (VA_SEQL) is typical of large image processing programs that are using virtual memory to sequentially address voluminous data, or <blank> (VA_NORM).

The final letter indicates the type of scheduling the process is using. If the process is executing using a fixed CPU schedule, the letter **F** is show; otherwise the field is blank.

UNIQUE OUTPUT FIELDS

Fields which are not common to all output formats:

USER Name of the owner of the process.

%CPU CPU utilization of the process; this is a decaying average of up to a minute of previous (real) time. Since the time base over which this is computed varies, since processes may be very young, it is possible for the sum of all %CPU fields to exceed 100%.

NICE (or NI) Process scheduling increment (see *setpriority*(2)).

SZ (or SIZE) Virtual size of the process in 1-Kbyte units. In the user-oriented output (the **u** option) this field includes the text segment while in the virtual memory statistics output (the **v** option) and in the long listing output (the **l** option) the text segment is not included.

RSS Real memory (resident set) size of the process in 1-Kbyte units.

LIM Soft limit on memory used, specified via a call to *setrlimit*(2); if no limit has been

specified, then it is shown as *xx*.

TSIZ Size of text (shared program) image.

TRS Size of resident (real memory) set of text.

%MEM Percentage of real memory used by this process.

RE Residency time of the process (seconds in core).

SL Sleep time of the process (seconds blocked).

PAGEIN Number of disk I/O's resulting from references by the process to pages not loaded in core.

UID Numerical user ID of process owner.

PPID Numerical ID of parent of process.

CP Short-term CPU utilization factor (used in scheduling).

PRI Process priority (nonpositive when in noninterruptible wait).

WCHAN Event on which process is waiting (an address in the system).

F Flags associated with process as in *<sys/proc.h>*.

A process that has exited and has a parent, but has not been waited for by the parent is marked <defunct>; a process which is blocked trying to exit is marked <exiting>; *ps* makes an educated guess as to the file name and arguments given when the process was created by examining memory or the swap area. The method is inherently somewhat unreliable, and in any event, a process is entitled to destroy this information; so the names cannot be counted on too much.

FILES

/vmunix	system namelist
/dev/mem	kernel memory
/dev/drum	swap device
/etc/ttys	searched to find tty names
/lib/ps/ps.ttys	saved tty data
*/lib/kernsyms/symdata_**	kernel symbol addresses

SEE ALSO

kill(1), w(1)

BUGS

Things change while *ps* is running; the picture it gives is only a close approximation to reality.

If a displayed value is wider than the field width allowed for it, the data for that process may not appear under the correct field headings. However, there will always be at least one space between fields.

Even if the **w** option is not used the output of **ps** can be wider than 80 columns. This is due to the fact that a displayed value can be wider than the field width allowed for it.

NAME

ptx – permuted index

SYNOPSIS

ptx [option] ... [input [output]]

DESCRIPTION

Ptx generates a permuted index to file *input* on file *output* (standard input and output default). It has three phases: the first does the permutation, generating one line for each keyword in an input line. The keyword is rotated to the front. The permuted file is then sorted. Finally, the sorted lines are rotated so the keyword comes at the middle of the page. *Ptx* produces output in the form:

.xx "tail" "before keyword" "keyword and after" "head"

where .xx may be an *nroff* or *troff*(1) macro for user-defined formatting. The *before keyword* and *keyword and after* fields incorporate as much of the line as will fit around the keyword when it is printed at the middle of the page. *Tail* and *head,* at least one of which is an empty string "", are wrapped-around pieces small enough to fit in the unused space at the opposite end of the line. When original text must be discarded, '/' marks the spot.

The following options can be applied:

–f Fold upper and lower case letters for sorting.

–t Prepare the output for the phototypesetter; the default line length is 100 characters.

–w *n* Use the next argument, *n*, as the width of the output line. The default line length is 72 characters.

–g *n* Use the next argument, *n*, as the number of characters to allow for each gap among the four parts of the line as finally printed. The default gap is 3 characters.

–o only
Use as keywords only the words given in the *only* file.

–i ignore
Do not use as keywords any words given in the *ignore* file. If the **–i** and **–o** options are missing, use */usr/lib/eign* as the *ignore* file.

–b break
Use the characters in the *break* file to separate words. In any case, tab, newline, and space characters are always used as break characters.

–r Take any leading nonblank characters of each input line to be a reference identifier (as to a page or chapter) separate from the text of the line. Attach that identifier as a 5th field on each output line.

The index for this manual was generated using *ptx*.

FILES

/usr/bin/sort
/usr/lib/eign

BUGS

Line length counts do not account for overstriking or proportional spacing.

NAME

pwd – working directory name

SYNOPSIS

pwd

DESCRIPTION

Pwd prints the pathname of the working (current) directory.

SEE ALSO

cd(1), csh(1), getwd(3)

BUGS

In *csh*(1) the command *dirs* is always faster (although it can give a different answer in the rare case that the current directory or a containing directory was moved after the shell descended into it).

NAME

quota – display disk usage and limits

SYNOPSIS

quota [**–v**] [user]

DESCRIPTION

quota displays users' disk usage and limits. Only the superuser may use the optional *user* argument to view the limits of users other than himself.

quota without options displays only warnings about mounted file systems where usage is over quota.

OPTIONS

–v display user's quotas on all mounted file systems where quotas exist.

FILES

quotas quota file at the file system root

SEE ALSO

quotactl(2), quotaon(8), edquota(8), rquotad(8c)

NAME

ranlib – convert archives to random libraries

SYNOPSIS

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.

ERRORS

Ranlib can return the following error messages:

'old format archive' - Old style archive format.

'not an archive' - not an archive created by ar.

'no symbol table' - there is no symbol table in the object module listed.

'mangled string table' - object module's string table is mangled, or not correctly formatted.

'old format .o file' - object module is not in SOFF format.

'would overwrite existing __.SYMDEF' - there already exists a file __.SYMDEF in the current directory.

'can't create temporary' - unable to create the temporary file.

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. The loader, *ld*, warns when the modification date of a library is more recent than the creation date of its dictionary; but this means that you get the warning even if you only copy the library.

NAME

rcp – remote file copy

SYNOPSIS

rcp [**–p**] file1 file2
rcp [**–p**] [**–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** option 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.

By default, the mode and owner of *file2* are preserved if it already existed; otherwise the mode of the source file modified by the *umask*(2) on the destination host is used. The **–p** option causes *rcp* to attempt to preserve (duplicate) in its copies the modification times and modes of the source files, ignoring the *umask*.

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 "rname@rhost" to use *rname* rather than the current user name on the remote host.

You can also specify a dash ('-') in place of *file1* or *file2*. When used in place of *file1*, *rcp* copies from *stdin* until end of file. When used in place of *file2*, *rcp* copies to *stdout*.

DIAGNOSTICS

The same messages that *cp*(1) produces. In addition, the following message may appear:

"rcp: %s not in hosts database"
Meaning: "%s" is not contained in /etc/hosts.

SEE ALSO

cp(1), ftp(1C), rsh(1C), rlogin(1C)

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. This results in an error: "rcp: protocol error ..." which includes the output from the sourced file.

The destination user and hostname may have to be specified as "rhost.rname" when the destination machine is running the 4.2BSD version of *rcp*.

NOTES

Use of *stdin* or *stdout* may not be supported by other vendors' version of *rcp*. When used with those systems, unpredictable results may occur.

rcp knows that files can be greater than two gigabytes in size, and is prepared to transfer them between systems. This may cause problems with other vendors' version of *rcp*.

rcp is an optional product; for more information, contact your CONVEX sales representative.

NAME

rcs – change RCS file attributes

SYNOPSIS

rcs [options] file ...

DESCRIPTION

Rcs creates new RCS files or changes attributes of existing ones. An RCS file contains multiple revisions of text, an access list, a change log, descriptive text, and some control attributes. For *rcs* to work, the caller's login name must be on the access list, except if the access list is empty, the caller is the owner of the file or the superuser, or the **-i** option is present.

Files ending in ',v' are RCS files, all others are working files. If a working file is given, *rcs* tries to find the corresponding RCS file first in directory ./RCS and then in the current directory, as explained in *co* (1).

The following options may be used however options may not be combined. For example, **rcs –q –u program** is valid but **rcs –qu program** is not valid.

–i creates and initializes a new RCS file, but does not deposit any revision. If the RCS file has no path prefix, *rcs* tries to place it first into the subdirectory ./RCS, and then into the current directory. If the RCS file already exists, an error message is printed.

–a*logins* appends the login names appearing in the comma-separated list *logins* to the access list of the RCS file.

–A*oldfile* appends the access list of *oldfile* to the access list of the RCS file.

–e[*logins*] erases the login names appearing in the comma-separated list *logins* from the access list of the RCS file. If *logins* is omitted, the entire access list is erased.

–c*string* sets the comment leader to *string*. The comment leader is printed before every log message line generated by the keyword Log during checkout (see *co*). This is useful for programming languages without multi-line comments. During *rcs* **-i** or initial *ci*, the comment leader is guessed from the suffix of the working file.

–l[*rev*] locks the revision with number *rev*. If a branch is given, the latest revision on that branch is locked. If *rev* is omitted, the latest revision on the trunk is locked. Locking prevents overlapping changes. A lock is removed with *ci* or *rcs* **-u** (see below).

–u[*rev*] unlocks the revision with number *rev*. If a branch is given, the latest revision on that branch is unlocked. If *rev* is omitted, the latest lock held by the caller is removed. Normally, only the locker of a revision may unlock it. Somebody else unlocking a revision breaks the lock. This causes a mail message to be sent to the original locker. The message contains a commentary solicited from the breaker. The commentary is terminated with a line containing a single '.' or control-D.

–L sets locking to *strict*. Strict locking means that the owner of an RCS file is not exempt from locking for checkin. This option should be used for files that are shared.

–U sets locking to non-strict. Non-strict locking means that the owner of a file need not lock a revision for checkin. This option should NOT be used for files that are shared.

–n*name*[:*rev*]
associates the symbolic name *name* with the branch or revision *rev*. *Rcs* prints an error message if *name* is already associated with another number. If *rev* is omitted, the symbolic name is deleted.

–N*name*[:*rev*]
same as **-n**, except that it overrides a previous assignment of *name*.

–o*range* deletes ("outdates") the revisions given by *range*. A range consisting of a single revision number means that revision. A range consisting of a branch number means the latest revision on that branch. A range of the form *rev1–rev2* means revisions *rev1* to *rev2* on the same branch, *–rev* means from the beginning of the branch containing *rev* up to and including *rev*, and *rev–* means from revision *rev* to the end of the branch containing *rev*. None of the outdated revisions may have branches or locks.

–q quiet mode; diagnostics are not printed.

–s*state*[:*rev*] sets the state attribute of the revision *rev* to *state*. If *rev* is omitted, the latest revision on the trunk is assumed; If *rev* is a branch number, the latest revision on that branch is assumed. Any identifier is acceptable for *state*. A useful set of states is *Exp* (for experimental), *Stab* (for stable), and *Rel* (for released). By default, *ci* sets the state of a revision to *Exp*.

–t[*txtfile*] writes descriptive text into the RCS file (deletes the existing text). If *txtfile* is omitted, *rcs* prompts the user for text supplied from the std. input, terminated with a line containing a single '.' or control-D. Otherwise, the descriptive text is copied from the file *txtfile*. If the **-i** option is present, descriptive text is requested even if **-t** is not given. The prompt is suppressed if the std. input is not a terminal.

LIMITS

The maximum limits on version numbers, deltas, and total revisions which the *rcs* system can handle are dependent upon operating system, filesystem, and architectural considerations. These limits should not become a factor under normal usage.

DIAGNOSTICS

The RCS file name and the revisions outdated are written to the diagnostic output. The exit status always refers to the last file checked out, and is 0 if the operation was successful, 1 otherwise.

FILES

The caller of the command must have read/write permission for the directory containing the RCS file and read permission for the RCS file itself. *Rcs* creates a semaphore file in the same directory as the RCS file to prevent simultaneous update. For changes, *rcs* always creates a new file. On successful completion, *rcs* deletes the old one and renames the new one. This strategy makes links to RCS files useless.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 0.11 ; Release Date: 91/10/16 .

SEE ALSO

ci (1), co (1), ident (1), rcsdiff (1), rcsmerge (1), rlog (1), rcsfile (5), sccstorcs (8).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

NAME

rcsdiff – compare RCS revisions

SYNOPSIS

rcsdiff [**-biwt**] [**-cefhn**] [**-r***rev1*] [**-r***rev2*] file ...

DESCRIPTION

Rcsdiff runs *diff* (1) to compare two revisions of each RCS file given. A file name ending in ',v' is an RCS file name, otherwise a working file name. *Rcsdiff* derives the working file name from the RCS file name and vice versa, as explained in *co* (1). Pairs consisting of both an RCS and a working file name may also be specified.

All options except **–r** have the same effect as described in *diff*(1).

If both *rev1* and *rev2* are omitted, *rcsdiff* compares the latest revision on the trunk with the contents of the corresponding working file. This is useful for determining what you changed since the last checkin.

If *rev1* is given, but *rev2* is omitted, *rcsdiff* compares revision *rev1* of the RCS file with the contents of the corresponding working file.

If both *rev1* and *rev2* are given, *rcsdiff* compares revisions *rev1* and *rev2* of the RCS file.

Both *rev1* and *rev2* may be given numerically or symbolically.

EXAMPLES

The command

```
rcsdiff f.c
```

runs *diff* on the latest trunk revision of RCS file f.c,v and the contents of working file f.c.

IDENTIFICATION

Author: Walter F. Tichy
Revision Number: 0.8 ; Release Date: 91/10/08 .

SEE ALSO

ci (1), co (1), diff (1), ident (1), rcs (1), rcsmerge (1), rlog (1), rcsfile (5), sccstorcs (1).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

NAME

rcsmerge – merge RCS revisions

SYNOPSIS

rcsmerge -r*rev1* [**-r***rev2*] [**-p**] file

DESCRIPTION

Rcsmerge incorporates the changes between *rev1* and *rev2* of an RCS file into the corresponding working file. If **-p** is given, the result is printed on the std. output, otherwise the result overwrites the working file.

A file name ending in ',v' is an RCS file name, otherwise a working file name. *Merge* derives the working file name from the RCS file name and vice versa, as explained in *co* (1). A pair consisting of both an RCS and a working file name may also be specified.

Rev1 may not be omitted. If *rev2* is omitted, the latest revision on the trunk is assumed. Both *rev1* and *rev2* may be given numerically or symbolically.

Rcsmerge prints a warning if there are overlaps, and delimits the overlapping regions as explained in *co -j*. The command is useful for incorporating changes into a checked-out revision.

EXAMPLES

Suppose you have released revision 2.8 of f.c. Assume furthermore that you just completed revision 3.4, when you receive updates to release 2.8 from someone else. To combine the updates to 2.8 and your changes between 2.8 and 3.4, put the updates to 2.8 into file f.c and execute

```
rcsmerge -p -r2.8 -r3.4 f.c >f.merged.c
```

Then examine f.merged.c. Alternatively, if you want to save the updates to 2.8 in the RCS file, check them in as revision 2.8.1.1 and execute *co -j*:

```
ci -r2.8.1.1 f.c
co -r3.4 -j2.8:2.8.1.1 f.c
```

As another example, the following command undoes the changes between revision 2.4 and 2.8 in your currently checked out revision in f.c.

```
rcsmerge -r2.8 -r2.4 f.c
```

Note the order of the arguments, and that f.c will be overwritten.

IDENTIFICATION

Author: Walter F. Tichy
Revision Number: 0.8 ; Release Date: 91/10/08 .

SEE ALSO

ci (1), co (1), merge (1), ident (1), rcs (1), rcsdiff (1), rlog (1), rcsfile (5), sccstorcs (1).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

BUGS

Rcsmerge does not work for files that contain lines with a single '.'.

NAME

rdiff – remote diff front end

SYNOPSIS

rdiff [*diff switch*] ... [*host:*]*file* [*host:*]*file*

DESCRIPTION

Rdiff is a front end for the *diff* utility that allows *diff* processing of files on remote machines. The syntax of *rdiff* is exactly the same as for *diff*, with two exceptions: the files specified must be plain files (not directories), and files on remote machines can be accessed by prepending the file name with "*host:*", where *host* is the host name of the machine that the file resides on.

Any *diff switch* arguments are passed directly to the *diff* utility. Some *diff* switches only apply to directories; it does not make sense to use these switches with *rdiff*, since *rdiff* does not accept directories.

Rdiff operates by using *rcp* to copy any files on remote machines, invoking *diff* on the copied files, then removing the copied files.

EXAMPLES

Compare */etc/passwd* on the current machine with */etc/passwd* on machine *m1*:

rdiff /etc/passwd m1:/etc/passwd

Compare */etc/passwd* on machines *m1* and *m2*:

rdiff m1:/etc/passwd m2:/etc/passwd

Compare */etc/passwd* on the current machine with */etc/passwd* on machine *m1*, ignoring blanks and tabs at the end of lines, and producing lines of context:

rdiff -b -c /etc/passwd m1:/etc/passwd

FILES

/tmp/rdiff*

SEE ALSO

diff(1), rcp(1C) (optional product)

DIAGNOSTICS

Rdiff returns a 10 if there was an error, otherwise the exit status of the *diff* is returned.

NOTE

Rdiff is an optional product; for more information, contact your CONVEX sales representative.

NAME

rdist – remote file distribution program

SYNOPSIS

rdist [**–nqbRhivwyu**] [**–t** *timeout*] [**–f** *distfile*] [**–d** *var=value*] [**–m** *host*] [*name ...*]

rdist [**–nqbRhivwy**] [**–t** *timeout*] **-c** *name ...* [login@]*host*[:*dest*]

DESCRIPTION

rdist is a program to maintain identical copies of files over multiple hosts. It preserves the owner, group, mode, and *mtime* of files if possible and can update programs that are executing. *rdist* reads commands from *distfile* to direct the updating of files or directories. If *distfile* is "–", the standard input is used. If no **–f** option is present, the program looks first for "*distfile*", then "*Distfile*" to use as the input. If no names are specified on the command line, *rdist* updates all of the files and directories listed in *distfile*. Otherwise, the argument is taken to be the name of a file to be updated or the label of a command to execute. If label and filenames conflict, it is assumed to be a label. These may be used together to update specific files using specific commands.

The **–c** option forces *rdist* to interpret the remaining arguments as a small *distfile*. The equivalent distfile is as follows.

```
( name ... ) -> [login@]host
        install   [dest] ;
```

Other options:

–d Define *var* to have *value*. The **–d** option is used to define or override variable definitions in the *distfile*. *value* can be the empty string, one name, or a list of names surrounded by parentheses and separated by tabs, spaces, or tabs and spaces.

–m Limit which machines are updated. Multiple **-m** arguments can be given to limit updates to a subset of the hosts listed the *distfile*.

–n Print the commands without executing them. This option is useful for debugging *distfile*.

–q Quiet mode. Files that are being modified are normally printed on standard output. The **–q** option suppresses this.

–R Remove extraneous files. If a directory is being updated, any files that exist on the remote host that do not exist in the master directory are removed. This is useful for maintaining truly identical copies of directories.

–h Follow symbolic links. Copy the file that the link points to rather than the link itself.

–i Ignore unresolved links. *rdist* normally try to maintain the link structure of files being transferred and warn the user if all the links cannot be found.

–v Verify that the files are up to date on all the hosts. Any files that are out of date will be displayed, but no files are changed nor any mail sent.

–w Whole mode. The whole filename is appended to the destination directory name. Normally, only the last component of a name is used when renaming files. This preserves the directory structure of the files being copied instead of flattening the directory structure. For example, renaming a list of files such as (*dir1/f1 dir2/f2*) to *dir3* would create files *dir3/dir1/f1* and *dir3/dir2/f2* instead of *dir3/f1* and *dir3/f2*.

–y Younger mode. Files are normally updated if their *mtime* and *size* (see *stat*(2)) disagree. The **–y** option causes *rdist* not to update files that are younger than the master copy. This can be used to prevent newer copies on other hosts from being replaced. A warning message is printed for files are newer than the master copy.

–t If the remote host does not respond, **–t** causes the local host to break the connection after *timeout* minutes.

–b Binary comparison. Perform a binary comparison and update files if they differ, rather than comparing dates and sizes.

–u Update files only, do not install files. A warning message is printed.

distfile contains a sequence of entries that specify the files to be copied, the destination hosts, and what operations to perform to do the updating. Each entry has one of the following formats.

<variable name> '=' <name list>
[label:] <source list> '->' <destination list> <command list>
[label:] <source list> '::' <time_stamp file> <command list>

The first format is used for defining variables. The second format is used for distributing files to other hosts. The third format is used for making lists of files that have been changed since some given date. The *source list* specifies a list of files or directories on the local host which are to be used as the master copy for distribution. The *destination list* is the list of hosts to which these files are to be copied. Each file in the source list is added to a list of changes if the file is out of date on the host that is being updated (second format) or the file is newer than the time stamp file (third format).

Labels are optional. They are used to identify a command for partial updates.

Newlines, tabs, and blanks are only used as separators and are otherwise ignored. Comments begin with "#" and end with a newline.

Variables to be expanded begin with "$" followed by one character or a name enclosed in curly braces (see the examples at the end).

The source and destination lists have the following format:

<name>

or

"(" <zero or more names separated by white-space> ")"

The shell meta-characters "[", "]", "{", "}", "*", and "?" are recognized and expanded (on the local host only) in the same way as *csh*(1). They can be escaped with a backslash. The "~" character is also expanded in the same way as *csh* but is expanded separately on the local and destination hosts. When the **–w** option is used with a filename that begins with "~", everything except the home directory is appended to the destination name. Filenames that do not begin with "/" or "~" use the destination user's home directory as the root directory for the rest of the filename.

The command list consists of zero or more commands of the following format:

"install" <options> opt_dest_name ";"
"notify" <name list>";"
"except" <name list>";"
"except_pat" <pattern list>";"
"special" <name list>string ";"

The *install* command is used to copy out-of-date files or directories. Each source file is copied to each host in the destination list. Directories are recursively copied in the same way. *opt_dest_name* is an optional parameter to rename files. If no *install* command appears in the command list or the destination name is not specified, the source filename is used. Directories in the path-name are created if they do not exist on the remote host. To help prevent disasters, a non-empty directory on a target host is never replaced with a regular file or a symbolic link.

However, under the "–R" option, a non-empty directory is removed if the corresponding filename is completely absent on the master host. The *options* are **–R, –h, –i, –v, –w, –y,** and **–b** and have the same semantics as options on the command line except they only apply to the files in the source list. The login name used on the destination host is the same as the local host unless the destination name is of the format "login@host". In this case, a file *.rhosts* containing the name of the local host and user must exist in the home directory of user whose login name is to be used.

The *notify* command is used to mail the list of files updated (and any errors that may have occured) to the listed names. If no "@" appears in the name, the destination host is appended to the name (e.g., name1@host, name2@host, ...).

The *except* command is used to update all of the files in the source list **except** for the files listed in *name list*. This is usually used to copy everything in a directory except certain files.

The *except_pat* command is like the *except* command except that *pattern list* is a list of regular expressions (see *ed*(1) for details). If one of the patterns matches some string within a filename, that file is ignored. Because "\" is a quote character, it must be doubled to become part of the regular expression. Variables are expanded in *pattern list* but not shell file pattern matching characters. To include a "$", it must be escaped with "\".

The *special* command is used to specify *sh*(1) commands that are to be executed on the remote host after the file in *name list* is updated or installed. If the *name list* is omitted, the shell commands are executed for every file updated or installed. The shell variable "FILE" is set to the current filename before executing the commands in *string*. *string* starts and ends with "" and can cross multiple lines in *distfile*. Multiple commands to the shell should be separated by ";". Commands are executed in the user's home directory on the host being updated. The *special* command can be used to rebuild private databases. after a program has been updated.

The following is a small example.

```
HOSTS = ( matisse root@arpa)

FILES = ( /bin /lib /usr/bin /usr/games
        /usr/include/{*.h,{stand,sys,vax*,pascal,machine}/*.h}
        /usr/lib /usr/man/man? /usr/ucb /usr/local/rdist )

EXLIB = ( Mail.rc aliases aliases.dir aliases.pag crontab dshrc
        sendmail.cf sendmail.fc sendmail.hf sendmail.st uucp vfont )

${FILES} -> ${HOSTS}
        install -R ;
        except /usr/lib/${EXLIB} ;
        except /usr/games/lib ;
        special /usr/lib/sendmail "/usr/lib/sendmail -bz" ;

srcs:
/usr/src/bin -> arpa
        except_pat ( \\.o\$ /SCCS\$ ) ;

IMAGEN = (ips dviimp catdvi)

imagen:
/usr/local/${IMAGEN} -> arpa
        install /usr/local/lib ;
        notify ralph ;
```

```
${FILES} :: stamp.cory
        notify root@cory ;
```

FILES

distfile	input command file
/tmp/rdist*	temporary file for update lists

SEE ALSO

sh(1), csh(1), stat(2)

DIAGNOSTICS

A complaint about mismatch of *rdist* version numbers may really stem from some problem with starting your shell (e.g., you are in too many groups).

BUGS

Source files must reside on the local host where *rdist* is executed.

There is no easy way to have a special command executed after all files in a directory have been updated.

Variable expansion only works for name lists; there should be a general macro facility.

rdist aborts on files that have a negative *mtime* (before Jan 1, 1970).

There should be a "force" option to allow replacement of non-empty directories by regular files or symlinks. A means of updating file modes and owners of otherwise identical files is also needed.

NOTES

rdist is an optional product; for more information, contact your CONVEX sales representative.

NAME

restart – restart execution of a process or a process family

SYNOPSIS

restart [**-CFiqtvwWXz**] [**-k**|**-K** *signo*] *chkpnt-file*

DESCRIPTION

restart restarts a process or process hierarchy from a set of checkpoint files created using *chkpnt*(1). If a process hierarchy is to be restarted, the checkpoint file named on the command line must correspond to the root process of the hierarchy.

By default, a SIGCONT signal (see *kill*(1)) is sent to the root process in the hierarchy once all the processes have been restarted. See the **-k** or **-K** options to specify alternate signals.

After a successful restart, the process attributes of the processes in the hierarchy are restored to their state at the time of the checkpoint. These attributes include process id, parent process id, process group id, and process group and session relationships.

All references to the controlling terminal in the restarted process hierarchy are set to the controlling terminal of *restart* if one exists.

The list of options to *restart* follows:

-C Copy data files from the checkpoint directory to the original pathname. If any of a process's data files are copied into the checkpoint directory at the time of checkpoint, this option will cause *restart* to replace the contents of the original files with the copies. This guarantees that they are returned to the stored state. This is useful for files that are accessed randomly, where any portion of the file may have changed since the checkpoint. Be aware that the contents of the files are overwritten with the checkpointed files, so modifications to the file since the time of checkpoint will be lost.

-F Force a restart of the process even if non-restartable conditions exist. This includes such things as the desired pid is in use or data files are inaccessible.

-i Invoke *restart* in interactive mode.

-k *signo*
Send the target process a signal after the restart has completed successfully. Signals are specified by giving either the signal name (e.g., CONT or SIGCONT) or signal number (e.g., 19).

-K *signo*
Like **-k** but sends the specified signal to all processes in a process hierarchy.

-q Quiet mode. Suppress warning messages that may be generated during a restart. Fatal error messages are still generated.

-t The process is started in a traced state. This option is used to support debuggers.

-v Provide verbose informational output on restart.

-w Wait for the restarted process or processes to complete. *restart* exits with the exit value of the restarted process. This is the default behavior for *restart* but is supplied for compatibility with other implementations.

-W Do not wait for the restarted process. With this option, *restart* does not *fork*(2) and wait for the process hierarchy to complete but instead becomes the root of the hierarchy itself. Do not use this option from a shell command line because it causes *restart* to change its process id. This will confuse most interactive shells, effectively hanging a login session.

-X Print debugging output.

-z Restart the processes in the stopped state.

NOTES

Some of the more common reasons a restart may fail are listed below. Consult the *Convex Checkpoint Restart Guide* for more information.

A pid needed by one of the restarted process is currently in use in the system.

One or more of the file pathnames needed for restart are inaccessible.

The restarted process's current directory no longer exists.

DIAGNOSTICS

restart exits with a 0 when the restart succeeds, or the value of *errno* if the restart fails.

NOTES

restart is capable of restarting processes which use files of greater than two gigabytes in size

SEE ALSO

chkpnt(1), pattach(2), chkpnt(3), restart(3)
Convex Checkpoint Restart Guide

NAME

rev – reverse lines of a file

SYNOPSIS

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

rlog – print log messages and other information about RCS files

SYNOPSIS

rlog [options] file ...

DESCRIPTION

Rlog prints information about RCS files. Files ending in ',v' are RCS files, all others are working files. If a working file is given, *rlog* tries to find the corresponding RCS file first in directory ./RCS and then in the current directory, as explained in *co* (1).

Rlog with no options prints the following information for each RCS file: full path name of the RCS file, working file name, head (i.e., the number of the latest revision on the trunk), access list, locks, symbolic names, suffix, total number of revisions, number of revisions selected for printing, and descriptive text. This is followed by entries for revisions in reverse chronological order for each branch. For each revision, *rlog* prints revision number, author, date/time, state, number of lines added/deleted (with respect to the previous revision), locker of the revision (if any), and log message. The options below restrict this output.

–h prints only RCS file name, working file name, head, access list, locks, symbolic names, and suffix.

–t prints the same as **-h**, plus the descriptive text.

–d*dates* prints information about revisions with a checkin date/time in the ranges given by the semicolon-separated list of *dates*. A range of the form *d1*<*d2* or *d2*>*d1* selects the revisions that were deposited between *d1* and *d2*, (inclusive). A range of the form <*d* or *d*> selects all revisions dated *d* or earlier. A range of the form *d*< or >*d* selects all revisions dated *d* or later. A range of the form *d* selects the single, latest revision dated *d* or earlier. The date/time strings *d*, *d1*, and *d2* are in the free format explained in *co* (1). Quoting is normally necessary, especially for < and >. Note that the separator is a semicolon.

–l[*lockers*] prints information about locked revisions. If the comma-separated list *lockers* of login names is given, only the revisions locked by the given login names are printed. If the list is omitted, all locked revisions are printed.

–r*revisions*
prints information about revisions given in the comma-separated list *revisions* of revisions and ranges. A range *rev1*–*rev2* means revisions *rev1* to *rev2* on the same branch, –*rev* means revisions from the beginning of the branch up to and including *rev*, and *rev*– means revisions starting with *rev* to the end of the branch containing *rev*. An argument that is a branch means all revisions on that branch. A range of branches means all revisions on the branches in that range.

–s*states* prints information about revisions whose state attributes match one of the states given in the comma-separated list *states*.

–w[*logins*] prints information about revisions checked in by users with login names appearing in the comma-separated list *logins*. If *logins* is omitted, the user's login is assumed.

Combinations of the options **-d**, **-l**, **-r**, **-s**, and **-w** print the intersection of the revisions selected by each option. For these options, *rlog* also prints the information provided by **-t**.

IDENTIFICATION

Author: Walter F. Tichy
Revision Number: 0.8 ; Release Date: 91/10/08 .

SEE ALSO

ci (1), co (1), ident(1), rcs (1), rcsdiff (1), rcsmerge (1), rcsfile (5), sccstorcs (1).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

NAME

rlogin – remote login

SYNOPSIS

rlogin *rhost* [**–ec**] [**–l** *username*] [**–8**] [**-L**]
rhost [**–ec**] [**–l** *username*] [**–8**] [**-L**]

DESCRIPTION

Rlogin connects your terminal on the current local host system *lhost* to the remote host system *rhost*.

Each host has a file */etc/hosts.equiv* that contains a list of *rhost*'s with which it shares account names. (The host names must be the standard names as described in *rsh*(1C).) When you *rlogin* as the same user on an equivalent host, you do not need to give a password. Each user may also have a private equivalence list in *.rhosts* 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 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.

Your remote terminal type is the same as your local terminal type (as given in your environment TERM variable). **The terminal or window size is also copied to the remote system if the server supports the option, and changes in size are reflected as well. 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 sequence of the form **<CR>~.** disconnects from the remote host, where **~** is the escape character. A sequence of the form **<CR>~^Z** (where **^Z** is the suspend character) will suspend the *rlogin* session. A sequence of the form **<CR>~^Y** (where **^Y** is the delayed-suspend character) will suspend the *send* portion of the *rlogin* session, but still allow output from the remote system.

A different escape character may be specified by the **–e** option. There is no space separating this option flag and the argument character.

The **–8** option permits full 8-bit bytes to be t**ransferred to the remote host at all times. By default, only the low-order 7 bits are sent, except when the remote side's stop and start characters are not ^S and ^Q. This option may be useful when using editors on the remote host.**

The argument **–L** allows the rlogin session to be run in litout mode (see *tty*(4)).

SEE ALSO

rsh(1C)

FILES

/etc/hosts.equiv	contains a list of remote hosts which share account names
*/usr/hosts/**	for *rhost* version of the command

BUGS

More of the environment should be propagated.

NOTES

Rlogin is an optional product; for more information, contact your CONVEX sales representative.

NAME

rm, rmdir – remove (unlink) files or directories

SYNOPSIS

rm [**–f**] [**–r**] [**–i**] [–] file ...

rmdir dir ...

DESCRIPTION

Rm removes the entries for one or more files from a directory. If an entry was the last link to the file, the file is destroyed. Removal of a file requires write permission in its directory, but neither read nor write permission on the file itself.

If a file has no write permission and the standard input is a terminal, its permissions are printed and a line is read from the standard input. If that line begins with 'y' the file is deleted, otherwise the file remains. No questions are asked and no errors are reported when the **–f** (force) option is given.

If a designated file is a directory, an error comment is printed unless the optional argument **–r** has been used. In that case, *rm* recursively deletes the entire contents of the specified directory, and the directory itself.

If the **–i** (interactive) option is in effect, *rm* asks whether to delete each file, and, under **–r**, whether to examine each directory.

The null option – indicates that all the arguments following it are to be treated as file names. This allows the specification of file names starting with a minus.

Rmdir removes entries for the named directories, which must be empty.

SEE ALSO

unlink(2), rmdir(2)

NAME

rmail – handle remote mail received via uucp

SYNOPSIS

rmail user ...

DESCRIPTION

Rmail interprets incoming mail received via *uucp*(1C), 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*(8).

Rmail is explicitly designed for use with *uucp* and *sendmail*.

SEE ALSO

binmail(1), uucp(1C), sendmail(8)

NAME

rsh – remote shell

SYNOPSIS

rsh host [**–l** username] [**–n**] command
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*(1C)) to the originating account; no provision is made for specifying a password with a command.

If you omit *command,* instead of executing a single command, you will be logged in on the remote host using *rlogin*(1C).

The **–n** option is used to redirect the input of *rsh* to */dev/null.*

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.*

The environment variables HOME, PATH, USER and SHELL are set to the correct values for the remote machine and passed in to the remote shell at startup.

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(1C)

BUGS

If you are using *csh*(1) and put a *rsh*(1C) 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 *emacs*() or *vi*(1)); use *rlogin*(1C).

Stop signals stop the local *rsh* process only; this is arguably wrong, but currently hard to fix.

NOTES

Rsh is an optional product; for more information, contact your CONVEX sales representative.

NAME

ruptime – show host status of local machines

SYNOPSIS

ruptime [**–a**] [**–r**] [**–l**] [**–t**] [**–u**] [**machine...**]

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 11 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. The **–r** flag reverses the sort order.

If *machine* is specified, then *ruptime* displays information for that machine only. *machine* may be repeated to specify a list of machines.

FILES

/usr/spool/rwho/whod.* data files

SEE ALSO

rwho(1C), rwhod(8c)

BUGS

There is a limitation in the rwhod protocol that for more that 40 users to be transmited. This means that ruptime reports any more users than this as "40+"

NOTES

ruptime is an optional product; for more information, contact your CONVEX sales representative.

NAME

rwho – who's logged in on local machines

SYNOPSIS

rwho [**–a**] [**–i**] [**–m**] [**name ...**]

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.

If *name* is specified, then rwho displays information for user *name* only. *name* may be repeated to specify a list of users.

The output of *rwho* is first sorted by user name, then by idle time if the **–i** flag is given, then by the machine name, and then by the tty number. If the **–m** flag is specified, idle times are displayed in minutes.

FILES

/usr/spool/rwho/whod.* information about other machines

SEE ALSO

ruptime(1C), rwhod(8C)

BUGS

This is unwieldy when the number of machines on the local net is large. Also if there are more than 40 users on a remote machine the remaining users information will not be displayed.

NOTES

Rwho is an optional product; for more information, contact your CONVEX sales representative.

NAME

sccstorcs – build RCS file from SCCS file

SYNOPSIS

sccstorcs [**–t**] [**–v**] *s.file ...*

DESCRIPTION

Sccstorcs builds an RCS file from each SCCS file argument. The deltas and comments for each delta are preserved and installed into the new RCS file in order. Also preserved are the user access list and descriptive text, if any, from the SCCS file.

The following flags are meaningful:

–t Trace only. Prints detailed information about the SCCS file and lists the commands that would be executed to produce the RCS file. No commands are actually executed and no RCS file is made.

–v Verbose. Prints each command that is run while it is building the RCS file.

FILES

For each s.*somefile*, *Sccstorcs* writes the files *somefile* and *somefile*,v which should not already exist. *Sccstorcs* will abort, rather than overwrite those files if they do exist.

SEE ALSO

ci (1), co (1), rcs (1).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

DIAGNOSTICS

All diagnostics are written to stderr. Non-zero exit status on error.

BUGS

Sccstorcs does not preserve all SCCS options specified in the SCCS file. Most notably, it does not preserve removed deltas, MR numbers, and cutoff points.

AUTHOR

Ken Greer

NAME

script – make typescript of terminal session

SYNOPSIS

script [**–a**] [file]

DESCRIPTION

Script makes a typescript of everything printed on your terminal. The typescript is written to *file*, or appended to *file* if the **–a** option is given. It can be sent to the line printer later with *lpr*. If no file name is given, the typescript is saved in the file *typescript*.

The script ends when the forked shell exits.

This program is useful when using a crt and a hard-copy record of the dialog is desired, as for a student handing in a program that was developed on a crt when hard-copy terminals are in short supply.

BUGS

script places **everything** in the log file. This is not what the naive user expects.

Since *script* does not make an entry in the */etc/utmp* file programs which use the *getlogin*(2) call might get confused.

NAME

sed – stream editor (non-interactive text editor)

SYNOPSIS

sed [**–n**] [**–e** script] [**–f** sfile] [file] ...

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**'s, 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.

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, '/regular expression/', in the style of *ed*(1) modified thus:

The escape sequence '\n' matches a newline embedded in the pattern space.

A command line with no addresses selects every pattern space.

A command line with one address selects each pattern space that matches the address.

A command line with two addresses selects the inclusive range from the first pattern space that matches the first address through the next pattern space that matches the second. (If the second address is a number less than or equal to the line number first selected, only one line is selected.) Thereafter the process is repeated, looking again for the first address.

Editing commands can be applied only to non-selected pattern spaces by use of the negation function '!' (below).

In the following list of functions the maximum number of permissible addresses for each function is indicated in parentheses.

An argument denoted *text* consists of one or more lines, all but the last of which end with '\' to hide the newline. Backslashes in text are treated like backslashes in the replacement string of an 's' command, and may be used to protect initial blanks and tabs against the stripping that is done on every script line.

An argument denoted *rfile* or *wfile* must terminate the command line and must be preceded by exactly one blank. Each *wfile* is created before processing begins. There can be at most 10 distinct *wfile* arguments.

(1) a\
text

Append. Place *text* on the output before reading the next input line.

(2) b *label*

Branch to the ':' command bearing the *label*. If *label* is empty, branch to the end of the script.

(2) c\
text

Change. Delete the pattern space. With 0 or 1 address or at the end of a 2-address range, place *text* on the output. Start the next cycle.

(2)d Delete the pattern space. Start the next cycle.

(2)D Delete the initial segment of the pattern space through the first newline. Start the next cycle.

(2)g Replace the contents of the pattern space by the contents of the hold space.

(2)G Append the contents of the hold space to the pattern space.

(2)h Replace the contents of the hold space by the contents of the pattern space.

(2)H Append the contents of the pattern space to the hold space.

(1)i\
text

Insert. Place *text* on the standard output.

(2)n Copy the pattern space to the standard output. Replace the pattern space with the next line of input.

(2)N Append the next line of input to the pattern space with an embedded newline. (The current line number changes.)

(2)p Print. Copy the pattern space to the standard output.

(2)P Copy the initial segment of the pattern space through the first newline to the standard output.

(1)q Quit. Branch to the end of the script. Do not start a new cycle.

(2)r *rfile*
Read the contents of *rfile*. Place them on the output before reading the next input line.

(2)s*/regular expression/replacement/flags*
Substitute the *replacement* string for instances of the *regular expression* in the pattern space. Any character may be used instead of '/'. For a fuller description see *ed*(1). *Flags* is zero or more of

g Global. Substitute for all nonoverlapping instances of the *regular expression* rather than just the first one.

p Print the pattern space if a replacement was made.

w *wfile* Write. Append the pattern space to *wfile* if a replacement was made.

(2)t *label*
Test. Branch to the ':' command bearing the *label* if any substitutions have been made since the most recent reading of an input line or execution of a 't'. If *label* is empty, branch to the end of the script.

(2)w *wfile*
Write. Append the pattern space to *wfile*.

(2)x Exchange the contents of the pattern and hold spaces.

(2)y*/string1/string2/*
Transform. Replace all occurrences of characters in *string1* with the corresponding character in *string2*. The lengths of *string1* and *string2* must be equal.

(2)! *function*
Don't. Apply the *function* (or group, if *function* is '{') only to lines *not* selected by the address(es).

(0): *label*
This command does nothing; it bears a *label* for 'b' and 't' commands to branch to.

(1)= Place the current line number on the standard output as a line.

(2){ Execute the following commands through a matching '}' only when the pattern space is selected.

(0) An empty command is ignored.

sed supports comments in scripts. Any line containing an option string of white space (blanks or tabs) followed by a '#' is a comment and will be ignored by *sed*. Furthermore, if the first comment line is of the form '#n' the *sed* script will be executed as if the **-n** flag were specified.

RESTRICTIONS

A *sed* script may contain at most 200 commands and must be no more than 10000 bytes long.

SEE ALSO

ed(1), grep(1), awk(1), lex(1)
"Sed--A Non-interactive Text Editor" in the *ConvexOS Tutorial Papers*

NAME

send – initiate a UUCP phone call to a neighboring site

SYNOPSIS

send *site* [**-d***dir*] [**-x***level*]

DESCRIPTION

send starts up *uucico*, which initiates a UUCP call to the specified site. The site must exist in the local L.sys information file.

The **-d** and **-x** arguments are described in *uucico*(8).

FILES

/usr/lib/uucp/L.sys -- connecting site information
/usr/spool/uucp/STST/site -- status of last outgoing call

BUGS

Anyone may initiate an outgoing call.

Call status is not reported back to the initiating terminal.

NAME

sh – shell, the standard command programming language

SYNOPSIS

sh [**–acefhiknrstuvx**] [args]

DESCRIPTION

sh is a command programming language that executes commands read from a terminal or a file. 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 *sigvec*(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).

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
The pathname of the 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.

Environment

The environment (see *environ*(7)) is a list of name-value pairs that is passed to an executed program in the same way as a normal argument list. The 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 these two lines are equivalent:

```
TERM=450 cmd
(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=gb 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; 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*.

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.

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.

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 SIGTERM is ignored (so that **kill 0** does not kill an interactive shell) and SIGINT is caught and ignored (so that **wait** is interruptible). In all cases, SIGQUIT is ignored by the shell.

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

EXIT STATUS

Errors detected by the shell, such as syntax errors, cause the shell to return a non-zero exit status. If the shell is being used non-interactively execution of the shell file is abandoned. Otherwise, the shell returns the exit status of the last command executed (see also the **exit** command above).

FILES

/etc/profile
$HOME/.profile

/tmp/sh*
/dev/null

SEE ALSO

cd(1), echo(1), login(1), pwd(1), test(1), umask(1), dup(2), exec(2), fork(2), pipe(2), signal(2), umask(2), wait(2), environ(7)

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

size – size of an object file

SYNOPSIS

size [object ...]

DESCRIPTION

For each object-file argument, *size* prints the (decimal) number of bytes required by the text, data, bss, and initialized common block segments. If more than one file is being examined, or at least one of either thread data or thread bss segments exist in an object-file, the number of bytes required by them are displayed as well. The sum of all segments is then displayed in hex and decimal. If no file is specified, **a.out** is used.

SEE ALSO

a.out(5)

NAME

sleep – suspend execution for an interval

SYNOPSIS

sleep time

DESCRIPTION

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

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

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

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

SEE ALSO

getitimer(2), alarm(3C), sleep(3)

BUGS

Time must be less than 2,147,483,647 seconds.

NAME

sniff – continually watch the end of a file

SYNOPSIS

sniff [**–n**] file

DESCRIPTION

Sniff behaves similarly to **tail -f .** It displays the end of a file, sleeps a second and then looks again. When the file grows, sniff shows the appended text.

The **–n** parameter specifies how many characters from the end of the file to start displaying. The default is to start at the beginning of the file.

Sniff runs forever or until killed, whichever comes first.

SEE ALSO

tail(1)

NAME

sod – standard object file dump utility

SYNOPSIS

sod *file* [**–l**] [**–f**] [**–n**] [**–t**] [**-shdr** { *name* or *N* or **all** }] ...

DESCRIPTION

sod displays standard format object and core files in a human-readable form. Object file formats are fully described in *a.out*(5). The SOFF style format for core files is outlined in *core*(5).

The *file* argument must be given. The other options are:

-l Turn *long* form printing on. Flags in the file header, the optional header, and in the section headers will be expanded to names to show what is set. The default is to print the flags as hexadecimal values only.

-f Print the file header and the optional header.

-n Print the symbol table (name list). This option does not make sense for core files.

-s Display a file section. If *name* is given then only the named section will de displayed. If *N* is used, where *N* is a non-negative integer, then only the *N*-th file section is displayed. If **all** is used, then all of the file sections will be displayed. The modifiers **h**, **d**, and **r** select the output of the display. With no modifiers, **-s** displays only the section header. If **h** is specified, then the section header will *not* be displayed. If **d** is specified, the section's raw data (for example, program text for the *text* section) is printed. If the **r** modifier is given, then the section's relocation information is printed; this modifier does not apply to core files. The **-s** option may be repeated to display many sections.

–t Display a table of contents of file sections. Each entry in the table gives the name of the section, its size, its flag word, the number of relocation entries for the section, and the alignment for the section. If the file has been stripped or if it is a core file, the names of the sections will be replaced with numeric identifiers which may be used to display a particular section via the **-s** commands.

If the symbol and string tables have been removed from *file* with *strip*(1), *sod* will not display or recognize section names. If the file is a core file, specific sections may be requested using logical mnemonic names, such as *context, tcontext, text, ttext,* etc. If there are multiple sections of the same type (which is very possible for core files), all of them will be displayed.

If no arguments are specified on the command line, *sod* will enter an interactive mode and prompt you for commands. All command line arguments have the same usage and meaning, but the **-**'s are not recognized. There are some additional options available in interactive mode:

h Display the help message. The message lists the valid options and what each option does.

? Displays the help message also.

c Print the full pathname of the current working file.

e Switch to a new *file*. In the prompt, *sod* displays the last component of the current working file pathname; when you change files, the prompt will likewise change.

l Toggle the *long* form on and off (it is initially off).

q Quit from *sod*.

SEE ALSO

strip(1), a.out(5), core(5)

NAME

soelim – eliminate .so's from nroff input

SYNOPSIS

soelim [file ...]

DESCRIPTION

Soelim reads the specified files or the standard input and performs the textual inclusion implied by the *nroff* directives of the form

.so somefile

when they appear at the beginning of input lines. This is useful since programs such as *tbl* do not normally do this; it allows the placement of individual tables in separate files to be run as a part of a large document.

An argument consisting of a single minus (–) is taken to be a file name corresponding to the standard input.

Note that inclusion can be suppressed by using '*(rq instead of '.', i.e.

'so /usr/lib/tmac.s

A sample usage of *soelim* would be

soelim exum?.n | tbl | nroff –ms | col | lpr

SEE ALSO

colcrt(1), more(1)

AUTHOR

William Joy

BUGS

The format of the source commands must involve no strangeness – exactly one blank must precede and no blanks follow the file name.

NAME

sort – sort or merge files

SYNOPSIS

sort [**–mubdfinrct***x*] [+*pos1* [–*pos2*]] ... [**–o** name] [**–T** directory] [name] ...

DESCRIPTION

Sort sorts lines of all the named files together and writes the result on the standard output. The name '–' means the standard input. If no input files are named, the standard input is sorted.

The default sort key is an entire line. Default ordering is lexicographic by bytes in machine collating sequence. The ordering is affected globally by the following options, one or more of which may appear.

b Ignore leading blanks (spaces and tabs) in field comparisons.

d 'Dictionary' order: only letters, digits and blanks are significant in comparisons.

f Fold upper case letters onto lower case.

i Ignore characters outside the ASCII range 040-0176 in nonnumeric comparisons.

n An initial numeric string, consisting of optional blanks, optional minus sign, and zero or more digits with optional decimal point, is sorted by arithmetic value. Option **n** implies option **b.**

r Reverse the sense of comparisons.

t*x* 'Tab character' separating fields is *x*.

The notation +*pos1* –*pos2* restricts a sort key to a field beginning at *pos1* and ending just before *pos2*. *Pos1* and *pos2* each have the form *m.n*, optionally followed by one or more of the flags **bdfinr,** where *m* tells a number of fields to skip from the beginning of the line and *n* tells a number of characters to skip further. If any flags are present they override all the global ordering options for this key. If the **b** option is in effect *n* is counted from the first nonblank in the field; **b** is attached independently to *pos2*. A missing *.n* means .0; a missing –*pos2* means the end of the line. Under the **–t***x* option, fields are strings separated by *x*; otherwise fields are nonempty nonblank strings separated by blanks.

When there are multiple sort keys, later keys are compared only after all earlier keys compare equal. Lines that otherwise compare equal are ordered with all bytes significant.

These option arguments are also understood:

c Check that the input file is sorted according to the ordering rules; give no output unless the file is out of sort. The input file is *not* sorted, but rather a message indicating the first item that causes disorder is printed. Therefore, it does not make sense to use **–o** in combination with this option.

m Merge only; the input files are already sorted.

o The next argument is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs. The **–c** option should not be used along with the **–o** option.

T The next argument is the name of a directory in which temporary files should be made.

u Suppress all but one in each set of equal lines. Ignored bytes and bytes outside keys do not participate in this comparison.

EXAMPLES

sort file foo by the contents from column 20 to column 30.

```
sort +20 –30 foo
```

Print in alphabetical order all the unique spellings in a list of words. Capitalized words differ from uncapitalized.

```
sort –u +0f +0 list
```

Print the password file (*passwd*(5)) sorted by user id number (the 3rd colon-separated field).

```
sort -t: +2n /etc/passwd
```

Print the first instance of each month in an already sorted file of (month day) entries. The options **-um** with just one input file make the choice of a unique representative from a set of equal lines predictable.

```
sort -um +0 -1 dates
```

FILES

/usr/tmp/stm*, /tmp/* first and second tries for temporary files

SEE ALSO

uniq(1), comm(1), rev(1), join(1)

DIAGNOSTICS

Comments and exits with nonzero status for various trouble conditions and for disorder discovered under option **-c**.

BUGS

Lines longer than 1024 characters (including newline) may be silently truncated. This happens if there are greater than 6400 lines (approx.) or greater than 100,000 characters (approx.) in the input file.

NAME

spell, spellin, spellout – find spelling errors

SYNOPSIS

spell [**–v**] [**–b**] [**–x**] [**–d** hlist] [**–s** hstop] [**–h** spellhist] [file] ...

spellin [list]

spellout [**–d**] list

DESCRIPTION

Spell collects words from the named documents, and looks them up in a spelling list. Words that neither occur among nor are derivable (by applying certain inflections, prefixes 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, tbl* and *eqn*(1) constructions.

Under the **–v** option, all words not literally in the spelling list are printed, and plausible derivations from spelling list words are indicated.

Under the **–b** option, British spelling is checked. Besides preferring *centre, colour, 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.

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

The auxiliary files used for the spelling list, stop list, and history file may be specified by arguments following the **–d**, **–s**, and **–h** options. The default files are indicated below. Copies of all output may be accumulated in the history file. The stop list filters out misspellings (e.g. thier=thy–y+ier) that would otherwise pass.

Two routines help maintain the hash lists used by *spell.* Both expect a set of words, one per line, from the standard input. *Spellin* combines the words from the standard input and the preexisting *list* file and places a new list on the standard output. If no *list* file is specified, the new list is created from scratch. *Spellout* looks up each word from the standard input and prints on the standard output those that are missing from (or present on, with option **–d**) the hashed *list* file. For example, to verify that *hookey* is not on the default spelling list, add it to your own private list, and then use it with *spell*,

```
echo hookey | spellout /usr/dict/hlista
echo hookey | spellin /usr/dict/hlista > myhlist
spell –d myhlist huckfinn
```

FILES

/usr/dict/hlist[ab]	hashed spelling lists, American & British, default for **–d**
/usr/dict/hstop	hashed stop list, default for **–s**
/dev/null	history file, default for **–h**
/tmp/spell.$$*	temporary files
/usr/lib/spell	

SEE ALSO

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

British spelling was done by an American.

spell considers words with multiple prefixes or suffixes to be correct if they have a correctly spelled root. For example, *spell* considers talkinging, prepretalk and pretalking as correct.

NAME

spline – interpolate smooth curve

SYNOPSIS

spline [option] ...

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., 349ff) has two continuous derivatives, and sufficiently many points to look smooth when plotted, for example by *graph*(1G).

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

$$y_0'' = ky_1'', \quad y_n'' = ky_{n-1}''$$

is set by the next argument, by 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(1G), plot(1G)

DIAGNOSTICS

When data is not strictly monotone in x, *spline* reproduces the input without interpolating extra points.

BUGS

A limit of 1000 input points is enforced silently.

NAME

split – split a file into pieces

SYNOPSIS

split [*–n*] [file [name]]

DESCRIPTION

Split reads *file* and writes it in *n*-line pieces (default 1000), as many as necessary, onto a set of output files. The name of the first output file is *name* with **aa** appended, and so on lexicographically. If no output name is given, **x** is default.

If no input file is given, or if – is given in its place, then the standard input file is used.

NAME

strings – find the printable strings in a object, or other binary, file

SYNOPSIS

strings [–] [**–o**] [*–number*] file ...

DESCRIPTION

Strings looks for ascii strings in a binary file. A string is any sequence of 4 or more printing characters ending with a newline or a null. Unless the – flag is given, *strings* only looks in the initialized data space of object files. If the **–o** flag is given, then each string is preceded by its offset in the file (in decimal). If the *–number* flag is given then number is used as the minimum string length rather than 4.

Strings is useful for identifying random object files and many other things.

SEE ALSO

od(1)

BUGS

The algorithm for identifying strings is extremely primitive

NAME

strip – remove debugger stabs or symbols and relocation bits

SYNOPSIS

strip [**–vs**] name ...

DESCRIPTION

With no command line flags, *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 –s command line flag causes strip to remove only debugger symbol table entries (stabs) from the output. In conjunction with –s, the –v flag causes a summary of symbol table entries and string table bytes before and after stripping to be written to standard output.

The effect of *strip* with no command flags is the same as use of the **–s** option of *ld*.

SEE ALSO

ld(1), a.out(5)

NAME

stty – set terminal options

SYNOPSIS

stty [-a] [-g] [option ...]

DESCRIPTION

stty sets certain terminal I/O options for the device that is the current standard output. Without arguments, it reports the settings of certain terminal options for the device that is the standard output; the settings are reported on the standard error.

More detailed information may be found in **termios**(4). Options in the last group are implemented using options in the previous groups. Note: many combinations of options make no sense, but no sanity checking is performed.

OPTIONS

–a Report all of the option settings.

everything Same as **–a** .

all Report all normally used options settings.

–g Report all current settings in a form that can be used as an argument to another **stty** command.

speed The terminal speed alone is printed on the standard output.

size The terminal (window) sizes are printed on the standard output, first rows and then columns.

Control Modes

[–]**parenb** Enable parity generation and detection. With a '–', disable parity checking.

[–]**parodd** Select odd parity. With a '–', select even parity.

cs5 cs6 cs7 cs8
Select character size.

[–]**hupcl** Hang up connection on last close. With a '–', do not hang up connection.

[–]**hup** Same as **hupcl**.

[–]**cstopb** Use two stop bits per character. With a '–', use one stop bit per character.

[–]**cread** Enable the receiver. With a '–', disable the receiver.

[–]**clocal** Assume a line without modem control. With a '–', assume a line with modem control.

[–]**crtscts** Use RTS/CTS flow control. (This is unimplemented.) With a '–', do not use RTS/CTS flow control.

Input Modes

[–]**ignbrk** Ignore break on input. With a '–', do not ignore a break on input.

[–]**brkint** Signal **SIGINT** on break. With a '–', do not signal.

[–]**ignpar** Ignore parity errors. With a '–', do not ignore parity errors.

[–]**parmrk** Mark parity errors With a '–', do not mark parity errors.

[–]**inpck** Enable input parity checking. With a '–', disable input parity checking.

[–]**istrip** Strip input characters to seven bits. With a '–', do not strip input characters.

[–]**inlcr** Map NEWLINE to RETURN on input. With a '–', do not map on input.

[–]**igncr** Ignore RETURN on input. With a '–', do not ignore RETURN on input.

[–]**icrnl** Map RETURN to NEWLINE on input. With a '–', do not map.

[–]**iuclc** Map upper-case alphabetics to lower case on input. (This is unimplemented.) With a '–', do not map.

[–]**ixon** Enable **START/STOP** output control. With a '–', disable output control. When enabled, output is stopped by sending a **STOP** character and started by sending a **START** character.

[–]**iflow** Same as **ixon**.

[–]**ixany** Allow any character to restart output. With a '–', only restart with a **START** character. Only enabled when **iexten** is also set.

[–]**decctlq** Same as **–ixany**.

[–]**ixoff** Request that the system send **START/STOP** characters when the input queue is nearly empty/full. With a '–', request that the system not send **START/STOP** characters.

[–]**itandem** Same as **ixoff**.

[–]**imaxbel** Request that the system send a BEL character to your terminal, and not to flush the input queue, if a character is received when the input queue is full. With a '–', request that it flush the input queue and not send a BEL character. Only enabled when **iexten** is also set.

Output Modes

[–]**opost** Post-process output. With a '–', do not post-process output; ignore all other output modes.

[–]**onlcr** Map NEWLINE to RETURN-NEWLINE on output. With a '–', do not map.

[–]**onlcrnl** Same as **onlcr**.

[–]**oxtabs** Expand TAB to spaces on output. With a '–', do not expand TAB .

[–]**tabs** Preserve TAB characters when printing. With a '–', expand TAB to spaces on output. (Provides the same functionality as **oxtabs**.)

[–]**onoeot** Deliver EOT to the terminal. With a '–', ignore EOT on output.

[–]**olcuc** Map lower-case alphabetics to upper case on output. (This is unimplemented.) With a '–', do not map.

Local Modes

[–]**isig** Enable the checking of characters against the special characters **INTR , QUIT**, **SUSP**, and **DSUSP**, With a '–', disable this checking.

[–]**icanon** Enable canonical input (**ERASE**, **KILL**, **WERASE**, and **RPRNT** processing). With a '–', disable canonical input.

[–]**iexten** Enable the checking of characters against the special characters **DSUSP**, **EOL2**, **QUOTE**, **ERASE2**, **LNEXT**, **FLUSHO**, **WERASE**, and **RPRNT**. With a '–', disable this checking. Also enables **echoctl**, **echoke**, **echoprt**, **imaxbel**, and **ixany** (i.e. these options have no effect unless **iexten** is also set).

[–]**xcase** Perform canonical upper/lower-case presentation. (This is unimplemented.) With a '–', do not perform canonical upper/lower-case presentation.

[–]**echo** Echo back every character typed. With a '–', do not echo back.

[–]**echoe** Echo the **ERASE** character as a sequence of BACKSPACE-SPACE-BACKSPACE. With a '–', echo the **ERASE** character as itself.

[–]**crtbs** Same as **echoe**.

[–]**crterase** Same as **echoe**.

[–]**echok** Echo NEWLINE after echoing a **KILL** character. With a '–', do not echo NEWLINE after echoing a **KILL** character.

[–]**echonl** Echo NEWLINE, even if **echo** is not set. With a '–', do not echo NEWLINE if **echo** is not set.

[–]**noflsh** Disable flush after **INTR** or **QUIT**. With a '–', enable flush.

[–]**tostop** Stop background jobs that attempt to write to the terminal. With a '–', allow background jobs to write to the terminal.

[–]**echoctl** Echo control characters as ^**x** (and delete as '^**?**'.) Print two BACKSPACE characters following the **EOF** character (default CTRL-D). With a '–', echo control characters as themselves. Only enabled when **iexten** is also set.

[–]**ctlecho** Same as **echoctl**.

[–]**echoprt** Echo erased characters backwards within '\' and '/'; used on printing terminals. With a '–', echo erased characters as indicated by **echoe**. Only enabled when **iexten** is also set.

[–]**prterase** Same as **echoprt**.

[–]**echoke** Echo the **KILL** character by erasing each character on the line as indicated by **echoprt** and **echoe**. With a '–', echo the **KILL** character as indicated by **echoctl** and **echok**. Only enabled when **iexten** is also set.

[–]**crtkill** Same as **echoke**.

Special Requests

new Switch to new line discipline. Enable SUSP, DSUSP, RPRNT, FLUSH, WERASE, and LNEXT special characters if disabled.

old Switch to old line discipline. Disable SUSP, DSUSP, RPRNT, FLUSH, WERASE, and LNEXT special characters.

newcrt Combination of **new** and **crt** .

0 Hang up phone line immediately.

50 75 110 134 150 200 300 600 1200 1800 2400 4800 9600 19200 exta 38400 extb
Set terminal baud rate to the number given, if possible. (Not all speeds are supported by all hardware interfaces.)

gspeed Set input speed to 300 baud and output speed to 9600 baud.

cr0 cr1 cr2 cr3
Select style of delay for RETURN characters.

nl0 nl1 nl2 nl3
Select style of delay for LINEFEED characters.

tab0 tab1 tab2 tab3
Select style of delay for horizontal TAB characters.

bs0 bs1 Select style of delay for BACKSPACE characters.

ff0 ff1 Select style of delay for form FORMFEED characters.

vt0 vt1 Select style of delay for vertical TAB characters.

[–]**tilde** Convert "~" to "`" on output (for Hazeltine terminals). With a '–', do not convert "~".

[–]**litout** Send output characters without any processing. With a '–', do normal output

processing, inserting delays, etc.

[–]**mdmbuf** Start/stop output on carrier transitions. (This is not implemented.) With a '–', return error if write attempted after carrier drops.

[–]**flusho** Output is being discarded usually because user hit control O (internal state bit). With a '–', output is not being discarded.

[–]**nohang** Don't send hangup signal if carrier drops. With a '–', send hangup signal to control process group when carrier drops.

[–]**pendin** Input is pending after a switch from non-canonical to canonical input mode and will be re-input when a read becomes pending or more input arrives (internal state bit). With a '–', input is not pending.

[–]**altwerase**
Use alternate word-erase algorithm where words consist of adjacent alphanumeric and underscore characters. With a '–', use standard word-erase algorithm with white-space delimited words.

Control Assignments

control-character c
Set *control-character* to *c*, where *control-character* is one of **erase**, **erase2**, **kill**, **intr**, **quit**, **eof**, **eol**, **eol2**, **start**, **stop**, **susp**, **dsusp**, **rprnt**, **flush**, **werase**, **lnext**, **quote**, or **disable**. If *c* is preceded by a caret (^), (escaped from the shell) then the value used is the corresponding CTRL character (for instance, '^**D**' is a CTRL-D); '^**?**' is interpreted as DEL and '^–' is interpreted as undefined. If *c* is preceded by '**M-**', the value used is the corresponding META character (i.e. the character with the high bit set). For example, '**M-x**' is a META-x, or hexadecimal value 0xF8. The octal, hexadecimal or decimal value of the character may also be given directly by preceding argument with ' 0 ', ' 0x ' , or using the digits 0-9, respectively. If *c* is specified as "u" or "undef" the *control-character* is set to undefined by setting it to the value of the **DISABLE** character.

min *i* Set the **MIN** value to *i*. The value of *i* may be specified in the same manner as the *control-character* described above.

time *i* Set the **TIME** value to *i*. The value of *i* may be specified in the same manner as the *control-character* described above.

rows *i* Set the recorded number of rows on the terminal to *i*.

columns *i* Set the recorded number of columns on the terminal to *i*.

cols *i* An alias for **columns** *i*.

Combination Modes

cooked Process the ERASE, ERASE2, WERASE, KILL, INTR, QUIT, EOF, EOL, EOL2, STOP, START, SUSP, DSUSP, RPRNT, FLUSH, and LNEXT, characters specially, and perform output post-processing.

evenp or **parity**
Enable **parenb** and **cs7**.

oddp Enable **parenb**, **cs7**, and **parodd**.

–parity, **–evenp**, or **–oddp**
Disable **parenb**, and set **cs8**.

even Enable **inpck** and disable **parodd**.

odd Enable **inpck** and **parodd**.

ek Set the **ERASE** and **KILL** characters to # and @.

sane Reset all modes to some reasonable values.

crt Set options for a CRT (**echoctl**, and, if >= 1200 baud, **echoe**, and **echoke**).

dec Set all modes suitable for Digital Equipment Corp. operating systems users (ERASE, KILL, and **INTR** characters to **^?**, **^U**, and **^C**, **decctlq**, and **crt**).

term Set all modes suitable for the terminal type **term**, where **term** is one of **tty33**, **tty37**, **vt05**, **tn300**, **ti700**, or **tek**.

[–]**raw** Enable raw input and output. With a '–', disable raw I/O. In raw mode, there is no special processing of the ERASE, ERASE2, WERASE, KILL, INTR, QUIT, EOF, EOL, EOL2, STOP, START, SUSP, DSUSP, RPRNT, FLUSH, nor **LNEXT** characters, nor is there any output post-processing.

[–]**cbreak** Set **isig** and unset **icanon**. With a '–', set both **isig** and **icanon**.

[–]**nl** Unset **icrnl**, **onlcr**. With a '–', set them.

[–]**lcase** Map upper case to lower case. With a '–', do not map case.

[–]LCASE Same as **lcase**.

SEE ALSO

ioctl(2), **termios**(4) **tty**(4)

NAME

style – analyze surface characteristics of a document

SYNOPSIS

style [**–ml**] [**–mm**] [**–a**] [**–e**] [**–l** num] [**–r** num] [**–p**] [**–P**] file ...

DESCRIPTION

Style analyzes the surface characteristics of the writing style of a document. It reports on readability, sentence length and structure, word length and usage, verb type, and sentence openers. Because *style* runs *deroff* before looking at the text, formatting header files should be included as part of the input. The default macro package **–ms** may be overridden with the flag **–mm.** The flag **–ml**, which causes **deroff** to skip lists, should be used if the document contains many lists of non-sentences. The other options are used to locate sentences with certain characteristics.

–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)

BUGS

Use of non-standard formatting macros may cause incorrect sentence breaks.

NAME

su – substitute user ID temporarily

SYNOPSIS

su [–] [**-f** username] [shell arguments or command]

DESCRIPTION

su demands the password of the specified *username*. If it is given correctly *su* changes to that *username* and invokes *username*'s default login shell as listed in the password file. *su* does not change the current directory nor the user environment except for the **HOME**, **SHELL**, and **USER** variables. (All of these are taken from *username*'s password file entry. See *environ*(7) for a description of the user environment). The new user ID stays in force until the shell exits.

If no *username* is specified 'root' is assumed. To remind the superuser of his/her extra privileges and responsibilities, the shell normally substitutes '#' for its usual prompt. The superuser can use *su* to any *username* without having to supply a password. You can pipe commands into the *su* command to be executed under a specified *username*. You will have to supply the password unless you are the superuser.

If the dash (-) argument is given, then the shell is called with the command name equal to "-su". For most shells (for certain */bin/sh* and */bin/csh*), if the first character of the command name is equal to '-', then the shell will perform a full login: the *.profile* or *.login* file will be sourced, and you will be placed in the home directory of the specified user.

If the **-f** flag is given to *su*, that flag is passed to the invoked shell, causing a fastlogin in the case of */bin/csh* (*i.e.*, *.cshrc* is NOT executed during startup).

Any other arguments given to *su* are passed along as arguments to the shell being invoked. Thus other shell arguments may be included, such as **-c 'cmd'** for the *csh*.

SEE ALSO

sh(1), csh(1)

BUGS

Local administrative rules cause restrictions to be placed on who can *su* to 'root', even when the root password is known.

NAME

sum – sum and count blocks in a file

SYNOPSIS

sum [-s] [file ...]

DESCRIPTION

Sum calculates and prints a 16-bit checksum for the named file(s), and also prints the number of blocks in the file(s). It is typically used to look for bad spots, or to validate a file(s) communicated over some transmission line.

The -s option when applied to executable files that have been stamped with activation keys will cause the checksum to be calculated in a system independent manner. If activation keys are not present in the executable file, the -s option will have no effect and sum will revert to its default checksum calculation.

SEE ALSO

wc(1)

DIAGNOSTICS

'Read error' is indistinguishable from end of file on most devices; check the block count. This means that if *sum* encounters a read error, it will interpret that error as an end of file signal and will assume that it has read the entire file. Therefore, check the block count to insure that *sum* read the entire file. To do this, you need to know how many bytes are in a block. This should be specified here. For example, if the block size is 4096 bytes and you are testing a file that is 12K, then the number of blocks returned by *sum* should be 3.

NAME

symorder – rearrange name list

SYNOPSIS

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 terminal tabs

SYNOPSIS

tabs [**–n**] [terminal]

DESCRIPTION

Tabs sets the tabs on a variety of terminals. Various terminal names given in *term*(7) are recognized; the default is, however, suitable for most 300 baud terminals. If the **–n** flag is present, the left margin is not indented as normal. If *terminal* is not specified, the contents of the TERM environment variable are used as a terminal type.

SEE ALSO

stty(1), term(7), tset(1)

BUGS

Tset(1) is generally used for this function.

NAME

tail – output the last part of a file

SYNOPSIS

tail [[[+ | -] **n**][**lbc**]] [**-r**] [**-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. *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. If no number is specified, -10 is used.

Specifying **r** causes tail to print lines from the end of the file in reverse order. The default for **r** is to print the entire file this way. Specifying **f** causes *tail* to not quit at end of file, but rather wait and try to read repeatedly in hopes that the file will grow. **r** and **f** can be specified alone as follows:

tail -r file

tail -f file

tail is able to accept files greater than two gigabytes as command line arguments.

SEE ALSO

dd(1), cat(1), more(1), head(1)

BUGS

Various kinds of anomalous behavior may happen with character special files.

NAME

talk, otalk – talk to another user

SYNOPSIS

talk person [ttyname]

otalk person

DESCRIPTION

Talk is a visual communication program which copies lines from your terminal to that of another user.

If you wish to talk to someone on your own machine, then *person* is just the person's login name. If you wish to talk to a user on another host, then *person* is of the form :

host!user or
host.user or
host:user or
user@host

though *user@host* is perhaps preferred.

If you want to talk to a user who is logged in more than once, the *ttyname* argument may be used to indicate the appropriate terminal name. Terminal names are specified by *ttyxxx* where *xxx* is the *tty* number.

When first called, it sends the message

Message from TalkDaemon@his_machine...
talk: connection requested by your_name@your_machine.
talk: respond with: talk your_name@your_machine

to the user you wish to talk to. At this point, the recipient of the message should reply by typing

talk your_name@your_machine

It doesn't matter from which machine the recipient replies, as long as his login-name is the same. Once communication is established, the two parties may type simultaneously, with their output appearing in separate windows. Typing control-L will cause the screen to be reprinted, while your erase, kill, and word-kill characters will work in talk as normal. To exit, just type your interrupt character; *talk* then moves the cursor to the bottom of the screen and restores the terminal.

Permission to talk may be denied or granted by use of the *mesg* command. (Talking is allowed by default.) Certain commands, in particular *nroff (1)* and *pr*(1) disallow messages in order to prevent messy output.

otalk is an version of talk using an older, obsolete version of the talk protocol. The protocol used by otalk is system architecture dependent and may not interoperate correctly with programs on non-convex systems using the otalk protocol.

FILES

/etc/hosts	to find the recipient's machine
/etc/utmp	to find the recipient's tty

SEE ALSO

mesg(1), who(1), mail(1), write(1)

NAME

tar – process tape archives

SYNOPSIS

tar –c[**bflvwhBI**] *block device* [-C *dirname*] *filename...*
tar –r[**bflvwhBI**] *block device* [-C *dirname*] [*filename...*]
tar –t[**fvBI**] *device*
tar –u[**bflvwhBI**] *block device*
tar –x[**fmovwpBI**] *device* [*filename...*]

DESCRIPTION

Tar reads and writes archive files which conform to the **Archive/Interchange File Format** specified in *IEEE Std. 1003.1-1988*.

Options

The following options are available:

–c Creates a new archive; writing begins at the beginning of the archive, instead of after the last file.

–r Writes named files to the end of the archive.

–t Lists the names of all of the files in the archive.

–u Causes named files to be added to the archive if they are not already there, or have been modified since last written into the archive. This implies the **–r** option.

–x Extracts named files from the archive. If a named file matches a directory whose contents had been written onto the archive, that directory is recursively extracted. If a named file in the archive does not exist on the system, the file is created with the same mode as the one in the archive, except that the set-user-id and get-group-id modes are not set unless the user has appropriate privileges.

If the files exist, their modes are not changed except as described above. The owner, group and modification time are restored if possible. If no *filename* argument is given, the entire contents of the archive is extracted. Note that if several files with the same name are in the archive, the last one will overwrite all earlier ones.

–b Causes *tar* to use the next argument on the command line as the blocking factor for tape records. The default is 20. This option should only be used with raw magnetic tape archives. The block size is determined automatically when reading tapes up to a blocking factor of 512.

–f Causes *tar* to use the next argument on the command line as the name of the archive instead of the default. The default is the tape device currently allocated (with *tpmount*(1)), or */dev/rmt8* if no tape devices are allocated. If - is specified as a filename *tar* writes to the standard output or reads from the standard input, whichever is appropriate for the options given. Thus, *tar* can be used as the head or tail of a pipeline.

–l Tells *tar* to report if it cannot resolve all of the links to the files being archived. If –l is not specified, no error messages are written to the standard output. This modifier is only valid with the **–c**, **–r** and **–u** options.

–m Tells *tar* not to restore the modification times. The modification time of the file will be the time of extraction. This modifier is invalid with the **–t** option.

–h Forces *tar* to follow symbolic links as if they were normal files or directories. Normally, *tar* does not follow symbolic links. This modifier is only valid with the -c, -r, and -u options.

–o Causes extracted files to take on the user and group identifier of the user running the program rather than those on the archive. This modifier is only valid with the **–x** option.

–v Causes *tar* to operate verbosely. Usually, *tar* does its work silently, but the **v**

modifier causes it to print the name of each file it processes, preceded by the option letter. With the **–t** option, **v** gives more information about the archive entries than just the name.

–w Causes *tar* to print the action to be taken, followed by the name of the file, and then wait for the user's confirmation. If a word beginning with **y** is given, the action is performed. Any other input means "no". This modifier is invalid with the **–t** option.

–p This option restores files to their original modes, ignoring the present *umask*(2). This modifier is only valid with the -x option.

–B Force input and output blocking to 20 blocks per record. This option was added so that tar can work across communications channel where the blocking may not be maintained.

–I Put the tape drive into 100 ips streaming mode. The default is 50 ips.

–C Cause *tar* to perform a chdir(2) to the directory name that follows. This allows multiple directories not related by a close common parent to be archived using short path names. For example, to archive files from /usr/include and from /etc, one might use:

tar c -C /usr include -C /etc .

FILES

/dev/tty used to prompt the user for information when the **–w** option is specified.

NOTES

tar is capable of archiving files and generating archives greater than two gigabytes in size. There is limitation of eight megabytes on the size of the file that *tar* can archive.

SEE ALSO

cpio(1), dd(1), find(1), pax(1), cpio(5), tar(5)

COPYRIGHT

AUTHOR

Mark H. Colburn
NAPS International
117 Mackubin Street, Suite 1
St. Paul, MN 55102
mark@jhereg.MN.ORG

Sponsored by **The USENIX Association** for public distribution.

NAME

tbl – format tables for nroff

SYNOPSIS

tbl [files] ...

DESCRIPTION

Tbl is a preprocessor for formatting tables for ***nroff (1)***. The input files are copied to the standard output, except for lines between .TS and .TE command lines, which are assumed to describe tables and are reformatted.

EXAMPLE

As an example, letting \t represent a tab (which should be typed as a genuine tab) the input

```
.TS
c s s
c c s
c c c
l n n.
Household Population
Town\tHouseholds
\tNumber\tSize
Bedminster\t789\t3.26
Bernards Twp.\t3087\t3.74
Bernardsville\t2018\t3.30
Bound Brook\t3425\t3.04
Branchburg\t1644\t3.49
Bridgewater\t7897\t3.81
Far Hills\t240\t3.19
.TE
```

yields

Household Population		
Town	Households	
	Number	Size
Bedminster	789	3.26
Bernards Twp.	3087	3.74
Bernardsville	2018	3.30
Bound Brook	3425	3.04
Branchburg	1644	3.49
Bridgewater	7897	3.81
Far Hills	240	3.19

If no arguments are given, *tbl* reads the standard input, so it may be used as a filter. When *tbl* is used with *neqn*, the *tbl* command should be first, to minimize the volume of data passed through pipes.

SEE ALSO

neqn(1)

"*TBL*– A Program to Format Tables" in the *ConvexOS Tutorial Papers.*

NAME

tee – pipe fitting

SYNOPSIS

tee [**–i**] [**–a**] [file] ...

DESCRIPTION

Tee duplicates the standard input, copying it to the standard output and to the specified *files*. Option **–i** ignores interrupts; option **–a** causes the output to be appended to the *files* rather than overwriting them.

EXAMPLE

% cat | tee file1 file2 | wc

This is going to file1 and file2, and being piped to 'wc'. <EOT>

NAME

tellcron – tell cron daemon to update its event list

SYNOPSIS

tellcron

DESCRIPTION

tellcron forces the *cron* daemon to update the invoking user's event list. The user's *.crontab* file is examined to see if it has been recently created or modified; in either case, the user's old commands (if any) are removed from the cron event list, and his new commands are added.

tellcron is particularly useful after modifying a *.crontab* file. Instead of modifying the event list at cron's next scheduled update, *tellcron* adds, removes, and reschedules a user's events immediately.

FILES

$HOME/.crontab - file of cron commands

SEE ALSO

cron(1)

NAME

telnet – User interface to the TELNET protocol

SYNOPSIS

telnet [**–d**] [**–n** *tracefile*] [*host* [*port*]]

DESCRIPTION

The **telnet** command is used to communicate with another host using the TELNET protocol. If **telnet** is invoked without the *host* argument, 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** will attempt to enable the TELNET LINEMODE option. If this fails, then **telnet** will revert to one of two input modes: either "character at a time" or "old line by line" depending on what the remote system supports.

When LINEMODE is enabled, character processing is done on the local system, under the control of the remote system. When input editing or character echoing is to be disabled, the remote system will relay that information. The remote system will also relay changes to any special characters that happen on the remote system, so that they can take effect on the local system.

In "character at a time" mode, most text typed is immediately sent to the remote host for processing.

In "old line by line" mode, all text is echoed locally, and (normally) only completed lines are sent to the remote host. The "local echo character" (initially "^E") may be used to turn off and on the local echo (this would mostly be used to enter passwords without the password being echoed).

If the LINEMODE option is enabled, or if the **localchars** toggle is TRUE (the default for "old line by line"; see below), the user's **quit**, **intr**, and **flush** characters are trapped locally, and sent as TELNET protocol sequences to the remote side. If LINEMODE has ever been enabled, then the user's **susp** and **eof** are also sent as TELNET protocol sequences, and **quit** is sent as a TELNET ABORT instead of BREAK. There are options (see **toggle autoflush** and **toggle autosynch** below) which cause this action to flush subsequent output to the terminal (until the remote host acknowledges the TELNET sequence) and flush previous terminal input (in the case of **quit** and **intr**).

While connected to a remote host, **telnet** command mode may be entered by typing the **telnet** "escape character" (initially "^]"). When in command mode, the normal terminal editing conventions are available. After many of the commands terminal returns to the connected mode automatically.

Options:

–d Sets the initial value of the **debug** toggle to TRUE.

–n *tracefile*
Opens *tracefile* for recording trace information. See the **set tracefile** command below.

host Indicates the official name, an alias, or the Internet address of a remote host.

port Indicates a port (address of an application). If this is not specified, the default **telnet** port is used. The port can be a number, or any tcp service from /etc/services.

The following **telnet** commands are available. Only enough of each command to uniquely identify it need be typed (this is also true for arguments to the **mode**, **set**, **toggle**, **unset**, **slc**, and **display** commands).

close
Close a TELNET session and return to command mode.

display [*argument...*]
Displays all, or some, of the **set** and **toggle** values (see below).

mode *type*

Type is one of several options, depending on the state of the TELNET session. The remote host is asked for permission to go into the requested mode. If the remote host is capable of entering that mode, the requested mode will be entered.

character

Disable the TELNET LINEMODE option, or, if the remote side does not understand the LINEMODE option, then enter "character at a time" mode.

line

Enable the TELNET LINEMODE option, or, if the remote side does not understand the LINEMODE option, then attempt to enter "old-line-by-line" mode.

isig (-isig)

Attempt to enable (disable) the TRAPSIG mode of the LINEMODE option. This requires that the LINEMODE option be enabled.

edit (-edit)

Attempt to enable (disable) the EDIT mode of the LINEMODE option. This requires that the LINEMODE option be enabled.

?

Prints out help information for the **mode** command.

open *host* [[-]*port*]

Open a connection to the named host. If 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**(5)) or an Internet address specified in the "dot notation" (see **inet**(3N)). When connecting to a non-standard port, **telnet** omits any automatic initiation of TELNET options. When the port number is preceded by a minus sign, the initial option negotiation is done. After establishing a connection, the **.telnetrc** in the users home directory is opened. Lines beginning with a **#** are comment lines. Blank lines are ignored. Lines that begin without whitespace are the start of a machine entry. The first thing on the line is the name of the machine that is being connected to. The rest of the line, and successive lines that begin with whitespace are assumed to be **telnet** commands and are processed as if they had been typed in manually to the **telnet** command prompt.

quit

Close any open TELNET session and exit **telnet**. An end of file (in command mode) will also close a session and exit.

send *arguments*

Sends one or more special character sequences to the remote host. The following are the arguments which may be specified (more than one argument may be specified at a time):

abort

Sends the TELNET ABORT (ABORT processes) sequence.

ao

Sends the TELNET AO (Abort Output) sequence, which should cause the remote system to flush all output **from** the remote system **to** the user's terminal.

ayt

Sends the TELNET AYT (Are You There) sequence, to which the remote system may or may not choose to respond.

brk

Sends the TELNET BRK (Break) sequence, which may have significance to the

remote system.

ec

Sends the TELNET EC (Erase Character) sequence, which should cause the remote system to erase the last character entered.

el

Sends the TELNET EL (Erase Line) sequence, which should cause the remote system to erase the line currently being entered.

eof

Sends the TELNET EOF (End Of File) sequence.

eor

Sends the TELNET EOR (End of Record) sequence.

escape

Sends the current **telnet** escape character (initially "^]").

ga

Sends the TELNET GA (Go Ahead) sequence, which likely has no significance to the remote system.

getstatus

If the remote side supports the TELNET STATUS command, **getstatus** will send the subnegotiation to request that the server send its current option status.

ip

Sends the TELNET IP (Interrupt Process) sequence, which should cause the remote system to abort the currently running process.

nop

Sends the TELNET NOP (No OPeration) sequence.

susp

Sends the TELNET SUSP (SUSPend process) sequence.

synch

Sends the TELNET SYNCH sequence. This sequence causes the remote system to discard all previously typed (but not yet read) input. This sequence is sent as TCP urgent data (and may not work if the remote system is a 4.2 BSD system -- if it doesn't work, a lower case "r" may be echoed on the terminal).

?

Prints out help information for the **send** command.

set *argument value*

unset *arguments...*

The **set** command will set any one of a number of **telnet** variables to a specific value or to TRUE. The special value **off** turns off the function associated with the variable, this is equivalent to using the **unset** command. The **unset** command will disable or set to FALSE any of the specified functions. The values of variables may be interrogated with the **display** command. The variables which may be set or unset, but not toggled, are listed here. In addition, any of the variables for the **toggle** command may be explicitly set or unset using the **set** and **unset** commands.

echo

This is the value (initially "^E") which, when in "line by line" mode, toggles between doing local echoing of entered characters (for normal processing), and suppressing echoing of entered characters (for entering, say, a password).

eof

If **telnet** is operating in LINEMODE or "old line by line" mode, entering this character as the first character on a line will cause this character to be sent to the remote system. The initial value of the eof character is taken to be the terminal's **eof** character.

erase

If **telnet** is in *localchars* mode (see **toggle localchars** below), **and** if **telnet** is operating in "character at a time" mode, then when this character is typed, a TELNET EC sequence (see **send ec** above) is sent to the remote system. The initial value for the erase character is taken to be the terminal's **erase** character.

escape

This is the **telnet** escape character (initially "^]") which causes entry into **telnet** command mode (when connected to a remote system).

flushoutput

If **telnet** is in *localchars* mode (see **toggle localchars** below) and the **flushoutput** character is typed, a TELNET AO sequence (see **send ao** above) is sent to the remote host. The initial value for the flush character is taken to be the terminal's **flush** character.

interrupt

If **telnet** is in *localchars* mode (see **toggle localchars** below) and the **interrupt** character is typed, a TELNET IP sequence (see **send ip** above) is sent to the remote host. The initial value for the interrupt character is taken to be the terminal's **intr** character.

kill

If **telnet** is in *localchars* mode (see **toggle localchars** below), **and** if **telnet** is operating in "character at a time" mode, then when this character is typed, a TELNET EL sequence (see **send el** above) is sent to the remote system. The initial value for the kill character is taken to be the terminal's **kill** character.

lnext If **telnet** is operating in LINEMODE or "old line by line" mode, then this character is taken to be the terminal's **lnext** character. The initial value for the lnext character is taken to be the terminal's **lnext** character.

quit

If **telnet** is in *localchars* mode (see **toggle localchars** below) and the **quit** character is typed, a TELNET BRK sequence (see **send brk** above) is sent to the remote host. The initial value for the quit character is taken to be the terminal's **quit** character.

reprint

If **telnet** is operating in LINEMODE or "old line by line" mode, then this character is taken to be the terminal's **reprint** character. The initial value for the reprint character is taken to be the terminal's **reprint** character.

start

If the TELNET TOGGLE-FLOW-CONTROL option has been enabled, then this character is taken to be the terminal's **start** character. The initial value for the kill character is taken to be the terminal's **start** character.

stop

If the TELNET TOGGLE-FLOW-CONTROL option has been enabled, then this character is taken to be the terminal's **stop** character. The initial value for the kill character is taken to be the terminal's **stop** character.

susp

If **telnet** is in **localchars** mode, or LINEMODE is enabled, and the **suspend** character is typed, a TELNET SUSP sequence (see **send susp** above) is sent to the remote host. The initial value for the suspend character is taken to be the terminal's **suspend** character.

tracefile

This is the file to which the output, caused by **netdata** or **option** tracing being TRUE, will be written. If it is set to '-', then tracing information will be written to standard output (the default).

worderase

If **telnet** is operating in LINEMODE or "old line by line" mode, then this character is taken to be the terminal's *worderase* character. The initial value for the worderase character is taken to be the terminal's *worderase* character.

slc *state*

The **slc** command (Set Local Characters) is used to set or change the state of the special characters when the TELNET LINEMODE option has been enabled. Special characters are characters that get mapped to TELNET commands sequences (like **ip** or **quit**) or line editing characters (like **erase** and **kill**). By default, the local special characters are exported.

export

Switch to the local defaults for the special characters. The local default characters are those of the local terminal at the time when **telnet** was started.

import

Switch to the remote defaults for the special characters. The remote default characters are those of the remote system at the time when the TELNET connection was established.

check

Verify the current settings for the current special characters. The remote side is requested to send all the current special character settings, and if there are any discrepancies with the local side, the local side will switch to the remote value.

?

Prints out help information for the **slc** command.

?

Displays the legal **set** (**unset**) commands.

toggle *arguments...*

Toggle (between TRUE and FALSE) various flags that control how **telnet** responds to events. These flags may be set explicitly to TRUE or FALSE using the **set** and **unset** commands listed above. More than one argument may be specified. The state of these flags may be interrogated with the **display** command. Valid arguments are:

autoflush

If **autoflush** and **localchars** are both TRUE, then when the **ao**, **intr**, or **quit** characters are recognized (and transformed into TELNET sequences; see **set** above for details), **telnet** refuses to display any data on the user's terminal until the remote system acknowledges (via a TELNET TIMING MARK option) that it has processed those TELNET sequences. The initial value for this toggle is TRUE if the terminal user had not done an "stty noflsh", otherwise FALSE (see **stty**(1)).

autosynch
If **autosynch** and **localchars** are both TRUE, then when either the **intr** or **quit** characters is typed (see **set** above for descriptions of the **intr** and **quit** characters), the resulting TELNET sequence sent is followed by the TELNET SYNCH sequence. This procedure **should** cause the remote system to begin throwing away all previously typed input until both of the TELNET sequences have been read and acted upon. The initial value of this toggle is FALSE.

binary
Enable or disable the TELNET BINARY option on both input and output.

inbinary
Enable or disable the TELNET BINARY option on input.

outbinary
Enable or disable the TELNET BINARY option on output.

crlf
If this is TRUE, then carriage returns will be sent as <CR><LF>. If this is FALSE, then carriage returns will be send as <CR><NUL>. The initial value for this toggle is FALSE.

crmod
Toggle carriage return mode. When this mode is enabled, most carriage return characters received from the remote host will be mapped into a carriage return followed by a line feed. This mode does not affect those characters typed by the user, only those received from the remote host. This mode is not very useful unless the remote host only sends carriage return, but never line feed. The initial value for this toggle is FALSE.

debug
Toggles socket level debugging (useful only to the *super*user). The initial value for this toggle is FALSE.

localchars
If this is TRUE, then the **flush**, **interrupt**, **quit**, **erase**, and **kill** characters (see **set** above) are recognized locally, and transformed into (hopefully) appropriate TELNET control sequences (respectively **ao**, **ip**, **brk**, **ec**, and **el**; see **send** above). The initial value for this toggle is TRUE in "old line by line" mode, and FALSE in "character at a time" mode. When the LINEMODE option is enabled, the value of **localchars** is ignored, and assumed to always be TRUE. If LINEMODE has ever been enabled, then **quit** is sent as **abort**, and **eof**and **suspend** are sent as **eof**and **susp**, see **send** above).

netdata
Toggles the display of all network data (in hexadecimal format). The initial value for this toggle is FALSE.

options
Toggles the display of some internal **telnet** protocol processing (having to do with TELNET options). The initial value for this toggle is FALSE.

prettydump
When the **netdata** toggle is enabled, if **prettydump** is enabled the output from the **netdata** command will be formated in a more user readable format. Spaces are put between each character in the output, and the beginning of any TELNET escape sequence is preceded by a '*' to aid in locating them.

?
Displays the legal **toggle** commands.

z

Suspend **telnet**. This command only works when the user is using the **csh**(1).

! [*command*]

Execute a single command in a subshell on the local system. If *command* is omitted, then an interactive subshell is invoked.

status

Show the current status of **telnet**. This includes the peer one is connected to, as well as the current mode.

? [*command*]

Get help. With no arguments, **telnet** prints a help summary. If a command is specified, **telnet** will print the help information for just that command.

FILES

~/.telnetrc

NOTES

On some remote systems, echo has to be turned off manually when in "old line by line" mode.

In "old line by line" mode or LINEMODE the terminal's *eof* character is only recognized (and sent to the remote system) when it is the first character on a line.

NAME

test, [– condition command

SYNOPSIS

test expr

DESCRIPTION

test evaluates the expression *expr*, and if its value is true returns with an exit status of zero; otherwise a non-zero exit status is returned. *test* returns a non-zero exit status if there are no arguments.

test is meant to be used with *sh* only, as similar functions are built into *csh*. These built-in commands are documented in the section covering expressions in *csh*(1).

To allow certain constructs in *sh* scripts, *test* may also be invoked as [. In this case, a closing] must terminate the expression; otherwise, *test* will generate an error message. An example of using *test* with [within a *sh* script is:

if [*expr*]

The following primitives are used to construct *expr*.

–r file true if the file exists and is readable.

–w file true if the file exists and is not a directory and is writable. A typical use for this is:

```
if test -w file
then
    command >> file          # append to file should always succeed
fi
```

–f file true if the file exists and is not a directory.

–d file true if the file exists and is a directory.

–s file true if the file exists and has a size greater than zero.

–l file true if the file exists and is a symbolic link.

–m file true if the file exists and is fully or partially migrated.

–p file true if the file exists and is a named pipe (fifo).

–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 nonzero.

s1 = s2 true if the strings *s1* and *s2* are equal.

s1 != s2 true if the strings *s1* and *s2* are not equal.

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**, or **–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

(expr)
parentheses for grouping.

–a has higher precedence than **–o**. Notice that all the operators and flags are separate arguments to *test*. In order to ensure correct parsing, all arguments should be separated by blanks. Notice also that parentheses are meaningful to the shell and must be escaped.

SEE ALSO

sh(1), find(1)

NAME

time – time a command

SYNOPSIS

time [**–e**] command

DESCRIPTION

The given command is executed; after it is complete, *time* prints the elapsed time during the command, the time spent in the system, and the time spent in execution of the command. Times are reported in seconds. The following option is interpreted by *time*.

–e Display times with extended accuracy. Extended accuracy implies whatever is available, up to micro-second resolution.

The times are printed on the diagnostic output stream.

Time is built in to *csh*(1), using a different output format. Use /bin/time with the *csh* to circumvent using the shell version.

BUGS

Elapsed time is accurate to the second, while the CPU times are measured to the 100th second. Thus the sum of the CPU times can be up to a second larger than the elapsed time.

NAME

tip, cu – connect to a remote system

SYNOPSIS

tip [**–v**] [*–speed*] system-name
tip [**–v**] [*–speed*] phone-number
cu phone-number [**–t**] [**–s** *speed*] [**-a** *acu*] [**–l** *line*] [*–n*]

DESCRIPTION

tip and *cu* establish a full-duplex connection to another machine, giving the appearance of being logged in directly on the remote CPU. However, you must have a login on the machine (or equivalent) to which you wish to connect. The preferred interface is *tip*. The *cu* interface is included for those people attached to the "call UNIX" command of version 7.

Typed characters are normally transmitted directly to the remote machine (which does the echoing as well). A tilde (~) appearing as the first character of a line is an escape signal; the following are recognized:

~^D ~. Drop the connection and exit (you may still be logged in on the remote machine).

~c [*name*] Change directory to name (no argument implies change to your home directory).

~! Escape to a shell (exiting the shell will return you to *tip*).

~> Copy file from local to remote. *tip* prompts for the name of a local file to transmit.

~< Copy file from remote to local. *tip* prompts first for the name of the file to be sent, then for a command to be executed on the remote machine.

~p *from* [*to*]
Send a file to a remote UNIX host. The put command causes the remote UNIX system to run the command string "cat > 'to'", while *tip* sends it the "from" file. If the "to" file isn't specified the "from" filename is used. This command is actually a UNIX specific version of the "~>" command.

~t *from* [*to*]
Take a file from a remote UNIX host. As in the "put" command, the "to" file defaults to the "from" filename if it isn't specified. The remote host executes the command string "cat 'from';echo ^A" to send the file to *tip*.

~| Pipe the output from a remote command to a local UNIX process. The command string sent to the local UNIX system is processed by the shell.

~# Send a BREAK to the remote system. For systems that don't support the necessary *ioctl* call, the break is simulated by a sequence of line speed changes and DEL characters.

~s Set a variable (see the discussion below).

~^Z Stop *tip* (only available with job control).

~? Get a summary of the tilde escapes

tip uses the file */etc/remote* to find how to reach a particular system and to find out how it should operate while talking to the system; refer to *remote*(5) for a full description. Each system has a default baud rate used to establish a connection. If this value is not suitable, the baud rate may be specified on the command line, e.g. "tip -300 mds".

When *tip* establishes a connection it sends out a connection message to the remote system; the default value, if any, is defined in */etc/remote*.

When *tip* prompts for an argument (e.g. during setup of a file transfer) the line typed may be edited with the standard erase and kill characters. A null line in response to a prompt, or an interrupt, will abort the dialogue and return you to the remote machine.

tip guards against multiple users connecting to a remote system by opening modems and terminal lines with exclusive access, and by honoring the locking protocol used by *uucp*(1C).

During file transfers *tip* provides a running count of the number of lines transferred. When using the ~> and ~< commands, the "eofread" and "eofwrite" variables are used to recognize end-of-file when reading, and specify end-of-file when writing (see below). File transfers normally depend on tandem mode for flow control. If the remote system does not support tandem mode, "echocheck" may be set to indicate *tip* should synchronize with the remote system on the echo of each transmitted character.

When *tip* must dial a phone number to connect to a system, it will print various messages indicating its actions. *tip* supports the DEC DN-11 and Racal-Vadic 831 auto-call-units; the DEC DF02 and DF03, Ventel 212+, Racal-Vadic 3451, and Bizcomp 1031 and 1032 integral call unit/modems.

tip can be aborted before a connection is made by typing the *quit* character, normally ^\ (control-backslash). After a connection is made, ~. or ~^D should be used to exit *tip*.

tip will look for two shell environment variables, PHONES and REMOTE. If PHONES is found, it contains the pathname of an alternate phone number data base. The default phone number data base is */etc/phones*. If REMOTE is found, and the value does not begin with a slash, the REMOTE remote string is used instead of an entry in */etc/remote*. If REMOTE does begin with a slash, the string is used as a pathname to the system descriptions file. The default system description file is */etc/remote*. If alternate files are specified in the PHONES or REMOTE environment variables, the referenced files should reside within a directory that is searchable by group; that is, the "group-executable" protection bit is set. The user must also be a member of the group that owns that directory.

One-line descriptions of *tip* usage are recorded in the log file */usr/adm/aculog*. This file must be present for *tip* to run. Because *tip* is a set-uid program to user *uucp*, this log file, along with the default system *remote* and *phones* files, should be owned by user *uucp*.

VARIABLES

tip maintains a set of *variables* that control its operation. Some of these variable are read-only to normal users (root is allowed to change anything of interest). Variables may be displayed and set through the "s" escape. The syntax for variables is patterned after *vi*(1) and *mail*(1). Supplying "all" as an argument to the set command displays all variables readable by the user. Alternatively, the user may request display of a particular variable by attaching a '?' to the end. For example "escape?" displays the current escape character.

Variables are numeric, string, character, or boolean values. Boolean variables are set merely by specifying their name; they may be reset by prepending a '!' to the name. Other variable types are set by concatenating an '=' and the value. The entire assignment must not have any blanks in it. A single set command may be used to interrogate as well as set a number of variables. Variables may be initialized at run time by placing set commands (without the "~s" prefix in a file *.tiprc* in one's home directory). The **–v** option causes *tip* to display the sets as they are made. Certain common variables have abbreviations. The following is a list of common variables, their abbreviations, and their default values.

beautify (bool) Discard unprintable characters when a session is being scripted; abbreviated *be*.

baudrate (num) The baud rate at which the connection was established; abbreviated *ba*.

dialtimeout (num) When dialing a phone number, the time (in seconds) to wait for a connection to be established; abbreviated *dial*.

echocheck (bool) Synchronize with the remote host during file transfer by waiting for the echo of the last character transmitted; default is *off*.

eofread (str) The set of characters which signify and end-of-transmission during a ~< file transfer command; abbreviated *eofr*.

eofwrite (str) The string sent to indicate end-of-transmission during a ~> file transfer command; abbreviated *eofw*.

eol (str) The set of characters which indicate an end-of-line. *tip* will recognize escape characters only after an end-of-line.

escape (char) The command prefix (escape) character; abbreviated *es*; default value is ~.

exceptions (str) The set of characters which should not be discarded due to the beautification switch; abbreviated *ex*; default value is "\t\n\f\b".

force (char) The character used to force literal data transmission; abbreviated *fo*; default value is ^**P**.

framesize (num) The amount of data (in bytes) to buffer between file system writes when receiving files; abbreviated *fr*.

host (str) The name of the host to which you are connected; abbreviated *ho*.

prompt (char) The character which indicates and end-of-line on the remote host; abbreviated *pr*; default value is **\n**. This value is used to synchronize during data transfers. The count of lines transferred during a file transfer command is based on receipt of this character.

raise (bool) Uppercase mapping mode; abbreviated *ra*; default value is *off*. When this mode is enabled, all lowercase letters will be mapped to uppercase by *tip* for transmission to the remote machine.

raisechar (char) The input character used to toggle uppercase mapping mode; abbreviated *rc*; default value is ^**A**.

record (str) The name of the file in which a session script is recorded; abbreviated *rec*; default value is "tip.record". SHELL is used to process the filename specified in *record* , so it is possible to use metacharacters, shell variables, and inline command substitution to generate the script filename.

script (bool) Session scripting mode; abbreviated *sc*; default is *off*. When *script* is *true*, *tip* will record everything transmitted by the remote machine in the script record file specified in *record*. If the *beautify* switch is on, only printable ASCII characters will be included in the script file (those characters between 040 and 0177). The variable *exceptions* is used to indicate characters which are an exception to the normal beautification rules.

tabexpand (bool) Expand tabs to spaces during file transfers; abbreviated *tab*; default value is *false*. Each tab is expanded to 8 spaces.

verbose (bool) Verbose mode; abbreviated *verb*; default is *true*. When verbose mode is enabled, *tip* prints messages while dialing, shows the current number of lines transferred during a file transfer operations, and more.

SHELL (str) The name of the shell to use for the ~! command; default value is "/bin/sh", or taken from the environment.

HOME (str) The home directory to use for the ~c command; default value is taken from the environment.

The following variables can also be set from the *.tiprc* file or with the ~**s** commands:

phones	remote	disconnect
tandem	linedelay	chardelay
etimeout	rawftp	halfduplex
localecho	parity	

CU INTERFACE

Phone-number is the telephone number, with minus signs at appropriate places for delays. The **–t** flag is used to dial out to a terminal. The **–s** flag sets the transmission speed to *speed*, 300 is the default value.

The **–a** and **–l** values may be used to specify pathnames for the ACU and communications line devices. They can be used to override the following built-in choices:

–a /dev/cua0 **–l** /dev/cul0

The *–n* option, where *n* is a single digit, changes the last character of the ACU and communications line to *n*. It is an abbreviation for **–a /dev/cua*n* –l /dev/cul*n*.**

FILES

/usr/adm/aculog	log file of *tip* usage, connections, failures
/etc/remote	global system descriptions
/etc/phones	global phone number data base
${REMOTE}	private system descriptions
${PHONES}	private phone numbers
~/.tiprc	initialization file
/usr/spool/uucp/LCK..*	lock file to avoid conflicts with *uucp*

DIAGNOSTICS

Diagnostics are self explanatory.

SEE ALSO

remote(5), phones(5)

BUGS

The full set of variables is undocumented.

NAME

touch – update date last modified of a file

SYNOPSIS

touch [**–a**] [**–c**] [**–f**] [**–m**] [**–s**] [**–l** *source_file*] [**mmddhhmm[yy]**] file ...

DESCRIPTION

touch attempts to set the modified date of each *file*. If a *file* exists, this is done using the *utimes* system call. If a *file* does not exist, an attempt will be made to create it unless the **–c** option is specified. The **–f** option will attempt to force the touch in spite of read and write permissions on a *file*. If **–a** is specified, only the *access* time of the file is updated. If **–m** is specified, only the *modification* time of the file is updated. If *file* is a symbolic link, *touch* attempts the set the modification date of the file referenced by the link. If the **–s** option is specified and *file* is a symbolic link, *touch* sets the modification date of the symbolic link itself.

If the option time string is specified, the times on the file are set according to the specified date and time.

If the **–l** option is specified, the times are set from the named *source_file* .

SEE ALSO

readlink(2), stat(2), symlink(2), utimes(2)

NAME

tpalloc, tpdealloc – tape drive allocation programs

SYNOPSIS

tpalloc [**-s**] [**-r**] [**-b**] [**-d** *bpi*] [**-i** *ips*] [**-f** *devname*] [*logicalname*] ...

tpdealloc *logicalname* ...
tpdealloc *devicename* ...

SPECIAL NOTE

This command was part of an older tape system. It is not part of the current tape system. There is, however, a shell script shipped with this release that accepts this command and converts it to the equivalent *tpmount* command. However, the default density is 1600 bpi and a link to the device is made in /tmp/TAPE<pid>.

This script will not be shipped with future releases of ConvexOS. You should learn the new tape commands as soon as possible.

DESCRIPTION

tpalloc and *tpdealloc* implement tape unit allocation. The *tpalloc* program is a request for exclusive access to a tape unit. *tpdealloc* relinquishes exclusive access of the tape unit.

If *logicalname* is specified, *tpalloc* attempts to make a symbolic link from a tape unit to *logicalname*. The characteristics of the tape unit can be specified by the **-d**, **-r**, **-b**, and **-i** options. The **-d** *bpi* option is used to specify the density of the tape unit (1600 bpi default). The **-r** option is used to request a tape unit which does not rewind when closed. The **-b** option is used to request a block mode tape unit. The **-i** *ips* option specifies a tape unit which spins the tape at *ips* inches per second.

tpalloc will attempt to allocate a tape unit with the requested characteristics. A specific tape unit can be requested by the **-f** *devname* option.

If the request is successful, *tpalloc* exits with status 0. It exits with status 1 if no resources were available, and exits with status 2 if an error occurred in the request. The **-s** option will cause *tpalloc* to suspend itself until the request is satisfied.

tpdealloc makes the tape unit, specified by *logicalname* or *devicename* available for others to allocate. By default, *tpdealloc* takes the tape unit offline as it deallocates the unit. The **-k** option is ignored. In the old tape system, it caused the tape to stay online.

SEE ALSO

tpmount(1), tpunmount(1), tpattr(1), tplabel(1), tpunlabel(1), tpqueue(1), tpwait(1), tape(3), mt(1), tpmnt(1), tpq(1), tprm(1), mtio(4)

NAME

tpattr – set attributes used for labelled tape files

SYNOPSIS

tpattr [**-s** *sname*] [**-b** *blocksize*] [**-r** *recordsize*] [**-f** *format*] [[**-h** *user_header_info*] ...] [[**-t** *user_trailer_info*] ...] [**-e** *expiration date*] [**-i** *file_identifier*] [**-a r|w|rw|""**]
[**-d blocknl|newline|syscall**]

DESCRIPTION

This command specifies labelled tape attributes for subsequent files written to the tape or read from the tape. These attributes must be set each time a tape is used. They are always set to their default values after a *tpmount*. Once an attribute is set, the attribute is applied to all files read/written until the attribute is again changed. Attributes must be set *before* the intended tape output file is opened. If incompatible attributes are chosen, the open of the labelled tape device will fail.

Options:

-s *sname*
: If you have only one mounted tape, the **-s** option is not required. Otherwise, the symbolic link name must be specified with the **-s** option.

-b *blocksize*
: Physical tape block size, in bytes. The default block size is 2048 bytes. The maximum block size allowed is 128 kilobytes.

-r *recordsize*
: Logical record size, in bytes. The default record size is 128 bytes.

-e *expirdate*
: Expiration date of the file. The format is DD-MM-[YY]YY. If a file is about to be overwritten and has not expired, the open(2) call will fail, setting errno to ENOACCESS. The default expiration date is January 1, 1900.

-f *format*
: Tape format. Recognized formats:

F - fixed length records
: When writing a tape, each logical record will be padded to the defined record length with spaces. When reading a tape, all spaces at the end of the record will be removed.

D - decimal coded variable length records.
: For use with ANSI labelled tapes only. When this format is selected, the recordsize specified with the **-r** option must take into account the four-byte record control word which ANSI requires to be written to tape for each logical record. For example, when using a record format of **D** and a logical record size of 512 bytes, there can really only be 508 bytes of actual data in each logical record, leaving room for the four-byte record control word in the logical record. When writing a tape, the label daemon will not pad a variable length record. When reading a tape, the label daemon will return only the data, the record control word is not returned. The application may use the byte count from the read(2) call as the record length.

V - binary coded variable length records.
: For use with IBM labelled tapes only. See decimal coded variable length records for an explanation of the restrictions of record length and block size.

VBS - variable length blocked and spanned records.
For use with IBM labelled tapes only. The label daemon will place the correct values into the header and trailer labels but will not block and span records. The tape system will treat this record format just like the undefined format. The application must block and span the records.

U - UNIX unformatted records.
When this format is selected, the record delimiter is ignored. ANSI does not define this format of records, therefore this type of record is not portable. IBM defines this record format. The record length field is ignored when this format is used. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

The default format is fixed.

-i *file_identifier* — The identifier for the file is contained in the HDR1 label. This identifier cannot be longer than 17 characters. The default file identifier is "NONAME".

-a r|w|rw|"" — Read/write access for users other than the owner. The owner always has read/write access. **r** grants read access to others, and **w** grants write access to others. The default access is **r**. The empty string "" grants neither read nor write access to others.

-d blocknl|newline|syscall — Logical record delimiter. **newline** indicates that logical tape records are split at newlines (<CTRL-J>, "\n"). **syscall** indicates that logical tape records are split on each *read*(2) or *write*(2) system call. **blocknl** is the same as **newline**, but a read gets as many logical records that will fit in the read buffer. **blocknl** is the default delimiter.

-h *user_header_info* — Information to be placed in the UHLn label on the tape. The user information is limited to seventy-six characters and is silently truncated. The option may be used up to 8 times. The first occurrence will relate to UHL1, the second to UHL2, and so forth. The access to the UHL information is through the tplist(1) command or by mounting the tape with bypass privileges and character mode and then reading the labels directly.

-t *user_trailer_info* — Information to be placed in the UTLn label on the tape. This information is independent of the user header labels.

SEE ALSO

tpmount(1), tpunmount(1), tplabel(1), tpunlabel(1), tpqueue(1), tpwait(1), tape(3), mt(1), ansidaemon(8), ibmdaemon(8)

NAME

tplabel – create a new labeled tape

SYNOPSIS

tplabel [**-s** *sname*] [**-a**]

DESCRIPTION

tplabel writes tape labels on a new or unlabeled tape. For this command to be used, the new tape must be mounted as labeled (see *tpmount*(1)). After the tape is labeled with *tplabel*, the tape can be used like any other labeled tape.

If you have only one tape mounted, the **-s** option is not required. Otherwise, the symbolic link name (given with *tpmount*) must be specified with the **-s** option.

If the **-a** option is specified, the new tape will have restricted access, and other users will not be able to read the tape. Ownership of a tape is determined by the owner field in the VOL1 header label. For CONVEX ANSI tapes, this field is set to the login name of the user that created the tape.

SEE ALSO

tpmount(1), tpunmount(1), tpattr(1), tpunlabel(1), tpqueue(1), tplist(1), tpwait(1), tape(3), mt(1)

NAME

tplist – list label information on a labelled tape

SYNOPSIS

tplist [**-d**] [**-f**] [**-r**] [**-v**] [**-s** *sname*]

DESCRIPTION

This command lists the contents of labels on a labelled tape. For labelled tapes, this command can show the owner of the tape, the accessibility, and information about the files on the tape.

The default output includes abbreviated information about the volume and the files. The options described below can be used to get more detailed information or suppress volume or file information.

Options:

-d Detailed printout.

-f Only show file labels. This option causes the suppression of volume labels, which are displayed by default.

-r Display raw labels. Label fields are normally separated out and printed in a human readable format. This option causes each label to be printed exactly as it is on the tape. A newline follows each raw label.

-v Only show volume labels. This option causes the suppression of file labels, which are displayed by default.

-s *sname* If you have only one mounted tape, the **-s** option is not required. Otherwise, the symbolic link name must be specified with the **-s** option.

After *tplist* is complete, the tape is positioned after the last file on the tape. If you want to read a file from the tape, the tape must be repositioned (see *mt*(1)). If you want to append a file to the end of the tape, no repositioning is needed.

SEE ALSO

tpmount(1), tpunmount(1), tplabel(1), tpunlabel(1), tpattr(1), tpqueue(1), tpwait(1), tape(3), mt(1), ansidaemon(8), ibmdaemon(8)

RESTRICTIONS

If *tplist* is interrupted by the user, the label daemon keeps reading the tape until it is positioned at the beginning of a file. If the file is large or if the file spans more than one volume, this may take a while.

NAME

tpmnt, tpumnt – tape mount request programs

SYNOPSIS

tpmnt [**-r**] [**-s**] [**-c** comment] [**-v** vsn] name

tpumnt name

SPECIAL NOTE

This command was part of an older tape system. It is not part of the current tape system. There is, however, a shell script shipped with this release that accepts this command and converts it to the equivalent *tpmount* command.

This script will not be shipped with future releases of ConvexOS. You should learn the new tape commands as soon as possible.

DESCRIPTION

tpmnt and *tpumnt* make requests for the tape operator to mount and dismount tapes. The **-r** option requests that the tape be read only, i.e the tape operator is to remove the write ring from the tape. The **-v** option requests that the tape with volume serial number *vsn* be mounted. *name* refers to either the logical name for the tape unit declared with *tpalloc* or the physical name for the tape unit.

The **-c** option will send the *comment* field parameter to the tape operator. If the tape operator is not present or the user has not allocated a tape unit with *tpalloc*, *tpmnt* will exit with status 1, otherwise, *tpmnt* will exit with status 0. The **-s** option will cause *tpmnt* to suspend until the tape is mounted, when the tape operator is not present. If **-s** isn't used, the request must be waited on with *tpwait* before using the tape.

tpumnt is a request for the tape operator to dismount the tape on the unit specified by *name*.

WARNINGS

In the old tape system, the drive had to be allocated with *tpalloc* before using *tpmnt* to request a tape mount. Now, *tpmnt* allocates the drive and asks for the tape mount; *tpalloc* must not be used first.

SEE ALSO

tpmount(1), tpunmount(1), tpattr(1), tplabel(1), tpunlabel(1), tpqueue(1), tpwait(1), tape(3), tpalloc(1), tpq(1), tprm(1)

NAME

tpmount – request tape mount or allocate a drive

SYNOPSIS

tpmount [**-a** *dev_name*] [**-B**] [**-**|**+b**] [**-c** *comment*] [**-d** *density*] [**-f** *N*] [**-i** *speed*] [**-l** *label_type*] [**-m** *mode*] [**-**|**+q**] [**-**|**+R**] [**-**|**+r**] [**-t** *drive_type*] [**-u**] [**-s** *sname* [*vsn,...*]]

DESCRIPTION

tpmount is used to request that a tape be mounted or a drive allocated. A drive is allocated without a tape being mounted when no VSN's are listed with the **-s** option.

Options:

-B Place the request in the background. The request must then be waited upon by *tpwait* before accessing the tape. A request can also be placed in the background by typing the keyboard interrupt character (usually <CTRL-C>) while *tpmount* is waiting for the request to complete.

The following options may be used more than once if several tapes must be mounted simultaneously. The order of these options is important. The **-s** option must appear *after* all other options for each tape.

-a *dev_name* Allocate the drive specified by the device *dev_name*.

-b, **+b** Bypass label processing. This allows a labelled tape to be used in the block or character device modes. This is a restricted option (see *tpconfig*(8)). **+b** de-selects this option.

-c *comment* A comment for the operator.

-d *density* Specifies the tape density.

-f *N* The number of files to skip after mounting the tape device. This option applies only to labelled tapes.

-i *speed* The speed of the device, as defined by *tpconfig*.

-l *label_type* Specifies the label type for labelled tapes. The only recognized types are **ansi** and **ibm**. There is no default value for labelled tapes; the user must define which type of label is desired.

-m *mode* Specifies the device mode, either *block*, *char*, or *label*. When *label* mode is used, if queueing is disabled the tape must be placed on the drive before issuing the *tpmount* command. The default value is defined within the tpconfig database.

-q, **+q** Causes *tpmount* to queue a drive allocation. This option only has meaning when queueing is disabled (see *tpconfig*(8)). Normally, if the requested drive is busy, *tpmount* immediately returns with the error message "no matching drives are available at this time". This option causes the allocation request to be queued within *tpdaemon* until it can be satisfied. **+q** de-selects this option.

-R, **+R** Tape should be read-only (mount tape without the write ring). **+R** de-selects this option.

-r, **+r** The rewind device is desired. This option applies only to tapes that are unlabeled or when label processing is by-passed. Labelled tapes are always assigned a no-rewind device. **+r** indicates that the no-rewind device is desired.

-s *sname* [*vsn,...*] *sname* is the name of a symbolic link which will point to the tape device. It is followed by a number of volume serial numbers. If multiple VSN's are given, they specify the volumes that make up one labelled fileset. Multiple VSN's are only allowed for labelled tapes.

Only the first six characters of a VSN are used for magnetic label verification. Extra characters may be used to identify rack location, etc.

If no VSN's are given, a tape drive is allocated without a tape being mounted. This is a restricted operation (see *tpconfig*(8)). If a drive is allocated without a tape, the user is responsible for mounting and unmounting his/her own tapes.

-t *drive_type* The drive type desired, as defined by the administrator with *tpconfig*. Examples: mt, dat, exabyte, 3480.

-u The internally buffered device is desired (for cartridge tapes only).

RESTRICTIONS

Labelled tapes are always assigned a no-rewind device.

Multiple tapes/drives can be mounted simultaneously by specifying the **-s** option for each drive, but in this release the system does not automatically prevent deadlocks when queueing is disabled (ie, when *opreq* isn't used). For example, if there are two tape drives and queueing is disabled, if two users each request both drives simultaneously, it is possible for each user to get one drive. When queueing is enabled, *opreq* prevents this kind of deadlock.

SEE ALSO

tpunmount(1), tpattr(1), tplabel(1), tpunlabel(1), tpqueue(1), tplist(1), tpwait(1), tape(3), mt(1), ansidaemon(8), ibmdaemon(8)

NAME

tpq – print tape request queue and tape unit utilization

SYNOPSIS

tpq [**-l**]

SPECIAL NOTE

This command was part of an older tape system. It is not part of the current tape system. There is, however, a shell script shipped with this release that accepts this command and converts it to the equivalent *tpqueue* command.

This script will not be shipped with future releases of ConvexOS. You should learn the new tape commands as soon as possible.

DESCRIPTION

tpq examines the tape request queue and prints its status on the standard output. *tpq*, also, prints the status of tape units, i.e. who has the unit allocated and what tape volume serial number is mounted on it. The **-l** causes a more detailed listing to be printed.

SEE ALSO

tpmount(1), tpunmount(1), tpattr(1), tplabel(1), tpunlabel(1), tpqueue(1), tpwait(1), tape(3), tpalloc(1), tpmnt(1), tprm(1)

NAME

tpqueue – print tape request queue and tape unit utilization

SYNOPSIS

tpqueue [**-l**]

DESCRIPTION

tpqueue prints the tape request queue on the standard output. The **-l** option causes a more detailed listing to be printed.

SEE ALSO

tpmount(1), tpunmount(1), tpattr(1), tplabel(1), tpunlabel(1), tplist(1), tpwait(1), tape(3), mt(1)

NAME

tprm – remove jobs from the tape mount request queue

SYNOPSIS

tprm

SPECIAL NOTE

This command was part of an older tape system. It is not part of the current tape system. There is, however, a shell script shipped with this release that accepts this command and converts it to the equivalent *tpunmount* command.

This script will not be shipped with future releases of ConvexOS. You should learn the new tape commands as soon as possible.

DESCRIPTION

tprm will remove a job from the tape mount request queue.

tprm in the old tape system accepted a job number or user name as an argument. This script ignores all arguments.

SEE ALSO

tpmount(1), tpunmount(1), tpattr(1), tplabel(1), tpunlabel(1), tpqueue(1), tpwait(1), tape(3), tpalloc(1), tpmnt(1), tpq(1)

NAME

tpunlabel – remove labels from a labeled tape

SYNOPSIS

tpunlabel [**-s** *sname*]

DESCRIPTION

tpunlabel erases labels from a labeled tape. For this command to be used, the labeled tape must be mounted as labeled (see *tpmount*(1)), and you must be the owner of the tape (or superuser). After the labels are erased, the only valid tape commands are *tpunmount* and *tplabel*. To use the unlabeled tape, it must be remounted as unlabeled (with *tpmount*).

If you have only one tape mounted, no options are required. Otherwise, the symbolic link name (given with *tpmount*) must be specified with the **-s** option.

SEE ALSO

tpmount(1), tpunmount(1), tpattr(1), tplabel(1), tpqueue(1), tplist(1), tpwait(1), tape(3), mt(1)

NAME

tpunmount – unmount a tape or deallocate a drive

SYNOPSIS

tpunmount [**-k**] [**-s** *sname*]

DESCRIPTION

tpunmount tells the tape system to unmount a tape. This command should be used when you are finished using a tape or drive.

If you have only one tape mounted (or one drive allocated), the **-s** option is not required. Otherwise, the symbolic link name (given with *tpmount*) must be specified with the **-s** option.

The **-k** option keeps the tape system from taking the tape offline. However, this option only has meaning when queueing is disabled (ie, *opreq* isn't being used). Also, once you unmount your tape, you have no control over what other users might do to the tape; someone else may take the tape offline, or even start using it. For these reasons, use of **-k** is discouraged. (This option exists only for backward compatibility with older tape commands.)

SEE ALSO

tpmount(1), tpattr(1), tplabel(1), tpunlabel(1), tpqueue(1), tpwait(1), tape(3), mt(1)

NAME

tpwait – wait for a *tpmount* to complete

SYNOPSIS

tpwait [**-s** *sname*]

DESCRIPTION

It is possible to put a tape mount request in the background with *tpmount* (see *tpmount*(1)). *tpwait* is used to wait for a background mount to complete.

If you have only one outstanding tape mount, no options are required. Otherwise, the symbolic link name (given with *tpmount*) must be specified with the **-s** option.

SEE ALSO

tpmount(1), tpunmount(1), tpattr(1), tplabel(1), tpunlabel(1), tpqueue(1), tape(3), mt(1)

NAME

tr – translate characters

SYNOPSIS

tr [**–cds**] [string1 [string2]]

DESCRIPTION

Tr copies the standard input to the standard output with substitution or deletion of selected characters. Input characters found in *string1* are mapped into the corresponding characters of *string2*. When *string2* is short it is padded to the length of *string1* by duplicating its last character. Any combination of the options **–cds** may be used: **–c** complements the set of characters in *string1* with respect to the universe of characters whose ASCII codes are 01 through 0377 octal; **–d** deletes all input characters in *string1;* **–s** squeezes all strings of repeated output characters that are in *string2* to single characters.

In either string the notation *a–b* means a range of characters from *a* to *b* in increasing ASCII order. The character '\' followed by 1, 2 or 3 octal digits stands for the character whose ASCII code is given by those digits. A '\' followed by any other character stands for that character.

The following example creates a list of all the words in 'file1' one per line in 'file2', where a word is taken to be a maximal string of alphabetics. The second string is quoted to protect '\' from the Shell. 012 is the ASCII code for newline.

```
tr –cs A–Za–z '\012' <file1 >file2
```

SEE ALSO

ed(1), ascii(7), expand(1)

BUGS

Won't handle ASCII NUL in *string1* or *string2;* always deletes NUL from input.

NAME

true, false – provide truth values

SYNOPSIS

true

false

DESCRIPTION

True and *false* are usually used in a Bourne shell script. They test for the appropriate status "true" or "false" before running (or failing to run) a list of commands.

EXAMPLE

```
while true
do
    command list
done
```

SEE ALSO

csh(1), sh(1), false(1)

DIAGNOSTICS

True has exit status zero.

NAME

tset – terminal dependent initialization
reset – reset the teletype bits to a sensible state

SYNOPSIS

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

reset ...

DESCRIPTION

Tset sets up your terminal when you first log in to a CONVEX system. It does terminal dependent processing such as setting erase and kill characters, setting or resetting delays, sending any sequences needed to properly initialized the terminal, and the like. It first determines the *type* of terminal involved, and then does necessary initializations and mode settings. The type of terminal attached to each ConvexOS port is specified in the */etc/ttys* database. Type names for terminals may be found in the *termcap*(5) database. If a port is not wired permanently to a specific terminal (not hardwired) it will be given an appropriate generic identifier such as *dialup*.

In the case where no arguments are specified, *tset* simply reads the terminal type out of the environment variable TERM and re-initializes the terminal. The rest of this manual concerns itself with mode and environment initialization, typically done once at login, and options used at initialization time to determine the terminal type and set up terminal modes.

When used in a startup script (*.profile* for *sh*(1) users or *.login* for *csh*(1) users) it is desirable to give information about the type of terminal you will usually use on ports which are not hardwired. These ports are identified in */etc/ttys* as *dialup* or *plugboard* or *arpanet*, etc. To specify what terminal type you usually use on these ports, the **–m** (map) option flag is followed by the appropriate port type identifier, an optional baud rate specification, and the terminal type. (The effect is to "map" from some conditions to a terminal type, that is, to tell *tset* "If I'm on this kind of port, guess that I'm on that kind of terminal".) If more than one mapping is specified, the first applicable mapping prevails. A missing port type identifier matches all identifiers. Any of the alternate generic names given in *termcap* may be used for the identifier.

A *baudrate* is specified as with *stty*(1), and is compared with the speed of the diagnostic output (which should be the control terminal). The baud rate *test* may be any combination of: >, @, <, and !; @ means "at" and ! inverts the sense of the test. To avoid problems with metacharacters, it is best to place the entire argument to **–m** within quotes; users of *csh*(1) must also put a "\" before any "!" used here.

Thus

tset –m 'dialup>300:adm3a' -m dialup:dw2 -m 'plugboard:?adm3a'

causes the terminal type to be set to an *adm3a* if the port in use is a dialup at a speed greater than 300 baud; to a *dw2* if the port is (otherwise) a dialup (i.e. at 300 baud or less). (**NOTE:** the examples given here appear to take up more than one line, for text processing reasons. When you type in real *tset* commands, you must enter them entirely on one line.) If the *type* finally determined by *tset* begins with a question mark, the user is asked if s/he really wants that type. A null response means to use that type; otherwise, another type can be entered which will be used instead. Thus, in the above case, the user will be queried on a plugboard port as to whether they are actually using an *adm3a*.

If no mapping applies and a final *type* option, not preceded by a **–m**, is given on the command line then that type is used; otherwise the identifier found in the */etc/ttys* database will be taken to be the terminal type. This should always be the case for hardwired ports.

It is usually desirable to return the terminal type, as finally determined by *tset*, and information about the terminal's capabilities to a shell's environment. This can be done using the – option; using the Bourne shell, *sh*(1):

export TERM; TERM=`tset – *options...*`

or using the C shell, *csh*(1):

setenv TERM `tset - *options...*`

With *csh* it is preferable to use the following command in your .login file to initialize the TERM and TERMCAP environment variables at the same time:

set noglob; eval "'tset -s *options...*'"

It is also convenient to make an alias in your .cshrc:

alias tset 'setenv TERM `/usr/ucb/tset - \!*`'

This allows the command

tset 2621

to be invoked at any time from your login csh to set the terminal and environment. **Note to Bourne Shell users:** It is **not** possible to get this aliasing effect with a shell script, because shell scripts cannot set the environment of their parent. (If a process could set its parent's environment, none of this nonsense would be necessary in the first place.)

These commands cause *tset* to place the name of your terminal in the variable TERM in the environment; see *environ*(7).

Once the terminal type is known, *tset* engages in terminal driver mode setting. This normally involves sending an initialization sequence to the terminal, setting the single character erase (and optionally the line-kill (full line erase)) characters, and setting special character delays. Tab and newline expansion are turned off during transmission of the terminal initialization sequence.

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

The options are:

–e*c* set the erase character to be the named character *c* on all terminals, the default being the backspace character on the terminal, usually ^H. The character *c* can either be typed directly, or entered using the hat notation used here.

–k*c* is similar to **–e** but for the line kill character rather than the erase character; *c* defaults to ^X (for purely historical reasons). The kill characters is left alone if **–k** is not specified. The hat notation can also be used for this option.

– The name of the terminal finally decided upon is output on the standard output. This is intended to be captured by the shell and placed in the environment variable TERM.

–s Print the sequence of *csh* commands to initialize the environment variables TERM and TERMCAP based on the name of the terminal finally decided upon.

–n On systems with the Berkeley 4BSD tty driver, specifies that the new tty driver modes should be initialized for this terminal. For a CRT, the CRTERASE and CRTKILL modes are set only if the baud rate is 1200 or greater. See tty(4) for more detail.

–I suppresses transmitting terminal initialization strings.

–Q suppresses printing the "Erase set to" and "Kill set to" messages.

–S prints the setting of the TERMCAP variable.

If **tset** is invoked as **reset**, it will set cooked and echo modes, turn off cbreak and raw modes, turn on newline translation, and restore special characters to a sensible state before any terminal dependent processing is done. Any special character that is found to be NULL or "**–1**" is reset to its default value.

This is most useful after a program dies leaving a terminal in a funny state. You may have to type "<LF>reset<LF>" to get it to work since <CR> may not work in this state. Often none of this will echo.

EXAMPLES

These examples all assume the Bourne shell and use the - option. If you use *csh*, use one of the variations described above. Note that a typical use of *tset* in a .profile or .login will also use the **–e** and **–k** options, and often the **–n** or **–Q** options as well. These options have not been included here to keep the examples small. (**NOTE:** some of the examples given here appear to take up more than one line, for text processing reasons. When you type in real *tset* commands, you must enter them entirely on one line.)

At the moment, you are on a 2621. This is suitable for typing by hand but not for a .profile, unless you are *always* on a 2621.

```
export TERM; TERM=`tset - 2621`
```

You have an h19 at home which you dial up on, but your office terminal is hardwired and known in /etc/ttys.

```
export TERM; TERM=`tset - -m dialup:h19`
```

You have a switch which connects everything to everything, making it nearly impossible to key on what port you are coming in on. You use a vt100 in your office at 9600 baud, and dial up to switch ports at 1200 baud from home on a 2621. Sometimes you use someone elses terminal at work, so you want it to ask you to make sure what terminal type you have at high speeds, but at 1200 baud you are always on a 2621. Note the placement of the question mark, and the quotes to protect the greater than and question mark from interpretation by the shell.

```
export TERM; TERM=`tset - -m 'switch>1200:?vt100' -m 'switch<=1200:2621'
```

All of the above entries will fall back on the terminal type specified in */etc/ttys* if none of the conditions hold. The following entry is appropriate if you always dial up, always at the same baud rate, on many different kinds of terminals. Your most common terminal is an adm3a. It always asks you what kind of terminal you are on, defaulting to adm3a.

```
export TERM; TERM=`tset - ?adm3a`
```

If the file */etc/ttys* is not properly installed and you want to key entirely on the baud rate, the following can be used:

```
export TERM; TERM=`tset - -m '>1200:vt100' 2621`
```

Here is a fancy example to illustrate the power of *tset* and to hopelessly confuse anyone who has made it this far. You dial up at 1200 baud or less on a concept100, sometimes over switch ports and sometimes over regular dialups. You use various terminals at speeds higher than 1200 over switch ports, most often the terminal in your office, which is a vt100. However, sometimes you log in from the university you used to go to, over the ARPANET; in this case you are on an ALTO emulating a dm2500. You also often log in on various hardwired ports, such as the console, all of which are properly entered in */etc/ttys*. You want your erase character set to control H, your kill character set to control U, and don't want *tset* to print the "Erase set to Backspace, Kill set to Control U" message.

```
export TERM; TERM=`tset -e -k^U -Q - -m 'switch<=1200:concept100' -m
'switch:?vt100' -m dialup:concept100 -m arpanet:dm2500`
```

FILES

/etc/ttys port name to terminal type mapping database
/etc/termcap terminal capability database

SEE ALSO

csh(1), sh(1), stty(1), ttys(5), termcap(5), environ(7)

AUTHORS

Eric Allman
David Wasley
Mark Horton

BUGS

The *tset* command is one of the first commands a user must master when getting started on the ConvexOS operating system. Unfortunately, it is one of the most complex, largely because of the extra effort the user must go through to get the environment of the login shell set. Something needs to be done to make all this simpler, either the *login*(1) program should do this stuff, or a default shell alias should be made, or a way to set the environment of the parent should exist.

NAME

tsort – topological sort

SYNOPSIS

tsort [file]

DESCRIPTION

Tsort produces on the standard output a totally ordered list of items consistent with a partial ordering of items mentioned in the input *file*. If no *file* is specified, the standard input is understood.

The input consists of pairs of items (nonempty strings) separated by blanks. Pairs of different items indicate ordering. Pairs of identical items indicate presence, but not ordering.

SEE ALSO

lorder(1)

DIAGNOSTICS

Odd data: there is an odd number of fields in the input file.

BUGS

Uses a quadratic algorithm; not worth fixing for the typical use of ordering a library archive file.

NAME

tty – get terminal name

SYNOPSIS

tty [-s]

DESCRIPTION

Tty prints the pathname of the user's terminal unless the –s (silent) is given. In either case, the exit value is zero if the standard input is a terminal and one if it is not.

DIAGNOSTICS

'not a tty' if the standard input file is not a terminal.

NAME

ul – do underlining

SYNOPSIS

ul [**–i**] [**–t** *terminal*] [*name ...*]

DESCRIPTION

Ul reads the named files (or standard input if none are given) and translates occurrences of underscores to the sequence which indicates underlining for the terminal in use, as specified by the environment variable TERM. The **–t** option overrides the terminal kind specified in the environment. The file */etc/termcap* is read to determine the appropriate sequences for underlining. If the terminal is incapable of underlining, but is capable of a standout mode then that is used instead. If the terminal can overstrike, or handles underlining automatically, *ul* degenerates to *cat*(1). If the terminal cannot underline, underlining is ignored.

The **–i** option causes *ul* to indicate underlining 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)

AUTHOR

Mark Horton wrote *ul*. The **–i** option was originally a option of the editor *ex*(1), then an *iul* command.

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

uniq – report repeated lines in a file

SYNOPSIS

uniq [**–udc** [+n] [–n]] [input [output]]

DESCRIPTION

uniq reads the specified input file (or standard input) comparing adjacent lines, and places the results in the output file or to the standard output. 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. When invoked with no options, *uniq* will print all unique lines and one copy of the duplicated lines.

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.

EXAMPLES

Assume there is a file *foo* that contains:

```
a
b
b
c
j
j
k
```

Then **uniq -d foo** will produce:

```
b
j
```

The command **uniq -u foo** will produce output consisting of:

```
a
c
k
```

Finally, with no options specified, **uniq foo** will give:

```
a
b
c
j
k
```

SEE ALSO

sort(1), comm(1)

BUGS

If *uniq* is invoked with both the **–d** and the **–u** flags, the output is *not* the same as *uniq* without options. *Uniq* will take the latter option as the overruling option, ignoring the first.

NAME

units – conversion program

SYNOPSIS

units [-][file]

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
* *2.54000e+00*
/ *3.93701e-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 pounds force/in2
You want: atm
* *1.02069e+00*
/ *9.79730e-01*

Units only does multiplicative scale changes. Thus it can convert Kelvin to Rankine, but not Centigrade 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 a unit of mass. Compound names are run together, e.g. 'lightyear'. British units that differ from their US counterparts are prefixed thus: 'brgallon'. Currency is denoted 'belgiumfranc', 'britainpound', ... A single dash causes ***units*** to display the entire ***units*** database. ***File*** is used as the ***units*** database instead of /usr/lib/units.

For a complete list of units, 'cat /usr/lib/units'.

FILES

/usr/lib/units

BUGS

Currency conversions are unreliable unless /usr/lib/units is maintained with up-to-date values.

NAME

uptime – show how long system has been up

SYNOPSIS

uptime

DESCRIPTION

Uptime prints the current time, the length of time the system has been up, the number of users logged in, and the average number of jobs in the run queue over the last 1, 5 and 15 minutes. It is, essentially, the first line of a *w*(1) command.

FILES

/vmunix system name list

SEE ALSO

w(1)

NAME

users – compact list of users who are on the system

SYNOPSIS

users

DESCRIPTION

Users lists the login names of the users currently on the system in a compact, one-line format.

FILES

/etc/utmp

SEE ALSO

who(1)

NAME

uucp – UNIX to UNIX copy

SYNOPSIS

uucp [**–acCdfmr**] [**–n***user*] [**–g***grade*] [**–s***spool*] [**–x***debug*] *source-file* [....] *destination-file*

DESCRIPTION

uucp copies files named by the *source-file* arguments to the *destination-file* argument. A filename may be a pathname on your machine, or may have the form

system-name!pathname

where *system-name* is taken from a list of system names that *uucp* knows about. Shell metacharacters ?*[] appearing in the pathname part are expanded on the appropriate system.

Pathnames may be one of:

(1) a full pathname

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

(3) a pathname prefixed by ~, where ~ is expanded into the system's public directory (usually */usr/spool/uucppublic*)

(4) a partial pathname, which 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*:

–a Avoid doing a *getwd* to find the current directory. (This is sometimes used for efficiency.)

–c Use the source file when copying out rather than copying the file to the spool directory. (This is the default.)

–C Copy the source file to the spool directory and transmit the copy.

–d Make all necessary directories for the file copy. (This is the default.)

–f Do not make intermediate directories for the file copy.

–g*grade*
grade is a single letter or number; lower ASCII sequence characters cause a job to be transmitted earlier during a particular conversation. Default is *n*. By way of comparison, *uux*(1C) defaults to *A*; mail is usually sent at *C*.

–m Send mail to the requester when the copy is complete.

–n*user* Notify *user* on remote system (i.e., send *user* mail) that a file was sent.

–r Do not start the transfer, just queue the job.

–s*spool* Use *spool* as the spool directory instead of the default.

–x*debug*
Turn on the debugging at level *debug*.

FILES

/usr/spool/uucp - spool directory
/usr/lib/uucp/* - other data and program files

SEE ALSO

uux(1C), mail(1)

D. A. Nowitz and M. E. Lesk, *A Dial-Up Network of UNIX Systems.*

D. A. Nowitz, *Uucp Implementation Description.*

WARNING

The domain of remotely accessible files can (and for obvious security reasons, usually should) be severely restricted. You will very likely not be able to fetch files by 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 the UUCP administrator (UID 14).

The **–m** option only works sending files or receiving a single file. (Receiving multiple files specified by special shell characters ?*[] does not activate the **–m** option.)

At present *uucp* cannot copy to a system several "hops" away, that is, a command of the form:

uucp myfile system1!system2!system3!yourfile

is not permitted. Use *uusend*(1C) instead.

When invoking *uucp* from *csh*(1), the "!" character must be prefixed by the "\" escape to inhibit *csh*'s history mechanism. (Quotes are not sufficient.)

uucp refuses to copy a file that does not give read access to "other"; that is, the file must have at least 0444 modes.

NAME

uuencode, uudecode – encode/decode a binary file for transmission via mail

SYNOPSIS

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*(1C) is not available.

uuencode takes the named source file (default is 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. The *remotedest* argument specifies the name of the file for 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 intent is that all mail to the user "decode" should be filtered through the *uudecode* program. This way the file is created automatically without human intervention. This is possible on the uucp network by either using *sendmail* or by making *rmail* be a link to *Mail* instead of *mail*. In each case, an alias must be created in a master file to get the automatic invocation of *uudecode*.

If these facilities are not available, the file can be sent to a user on the remote machine who can uudecode it manually.

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

atob(n), uusend(1C), uucp(1C), uux(1C), mail(1), uuencode(5)

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

uuencode, uudecode – encode/decode a binary file for transmission via mail

SYNOPSIS

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*(1C) is not available.

uuencode takes the named source file (default is 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. The *remotedest* argument specifies the name of the file for 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 intent is that all mail to the user "decode" should be filtered through the *uudecode* program. This way the file is created automatically without human intervention. This is possible on the uucp network by either using *sendmail* or by making *rmail* be a link to *Mail* instead of *mail*. In each case, an alias must be created in a master file to get the automatic invocation of *uudecode*.

If these facilities are not available, the file can be sent to a user on the remote machine who can uudecode it manually.

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

atob(n), uusend(1C), uucp(1C), uux(1C), mail(1), uuencode(5)

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 – display UUCP log files

SYNOPSIS

uulog **–s** *sys* | **–u** *user*

DESCRIPTION

uulog queries a log of *uucp*(1C) and *uux*(1C) transactions in the file */usr/spool/uucp/LOGFILE.*

The options command *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/LOGFILE

SEE ALSO

uucp(1C), uux(1C).

NOTES

Very early releases of UUCP used separate log files for each of the UUCP utilities; *uulog* was used to merge the individual logs into a master file. This capability has not been necessary for some time and is no longer supported.

BUGS

UUCP's recording of which user issued a request is unreliable.

uulog is little more than an overspecialized version of *grep*(1).

NAME

uuname – list names of UUCP hosts

SYNOPSIS

uuname [**–l**]

DESCRIPTION

uuname lists the UUCP names of known systems. The **–l** option returns the local system name; this may differ from the *hostname*(1) for the system if the *hostname* is very long.

SEE ALSO

uucp(1C), uux(1C).

NAME

uuq – examine or manipulate the uucp queue

SYNOPSIS

uuq [**-l**] [**-h**] [**-s***system*] [**-u***user*] [**-d***jobno*] [**-r***sdir*] [**-b***baud*]

DESCRIPTION

uuq is used to examine (and possibly delete) entries in the uucp queue.

When listing jobs, *uuq* uses a format reminiscent of *ls*. For the long format, information for each job listed includes job number, number of files to transfer, user who spooled the job, number of bytes to send, type of command requested (*S* for sending files, *R* for receiving files, *X* for remote uucp), and file or command desired.

Several options are available:

–h Print only the summary lines for each system. Summary lines give system name, number of jobs for the system, and total number of bytes to send.

–l Specifies a long format listing. The default is to list only the job numbers sorted across the page.

–s*system* Limit output to jobs for systems whose system names begin with *system*.

–u*user* Limit output to jobs for users whose login names begin with *user*.

–d*jobno* Delete job number *jobno* (as obtained from a previous *uuq* command) from the uucp queue. Only the UUCP Administrator is permitted to delete jobs.

–r*sdir* Look for files in the spooling directory *sdir* instead of the default directory.

–b*baud* Use *baud* to compute the transfer time instead of the default 1200 baud.

FILES

/usr/spool/uucp/	Default spool directory
/usr/spool/uucp/C./C.*	Control files
/usr/spool/uucp/D*hostname*./D.*	Outgoing data files
/usr/spool/uucp/X./X.*	Outgoing execution files

SEE ALSO

uucp(1C), uux(1C), uulog(1C), uusnap(8C)

BUGS

No information is available on work requested by the remote machine.

The user who requests a remote *uucp* command is unknown.

uuq -l can be horrendously slow.

NAME

uusend – send a file to a remote host

SYNOPSIS

uusend [**–m** *mode*] [**–r**] *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 is taken from the octal number given. Otherwise, the mode of the input file is used.

If the **-r** option is specified, the file transfer is queued rather that started immediately.

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.

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 should not 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

uux – unix to unix command execution

SYNOPSIS

uux [**-**] [**-cClLnprz**] [**-a***name*] [**-g***grade*] [**-x***debug*] command-string

DESCRIPTION

uux gathers zero or more files from various systems, executes a command on a specified system, and sends standard output to a file on a specified system.

The *command-string* is made up of one or more arguments that look like a Shell command line, except that the command and filenames 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 ~*user* where *user* is a login name on the specified system and is replaced by that user's login directory;

(3) a path name prefixed by ~; where ~ is expanded to the system's public directory (usually */usr/spool/uucppublic*);

(4) a partial pathname, which is prefixed by the current directory.

As an example, the command

uux "!diff convex!/mnt/jthomp/file1 byu!/a/jimt/file2 > !~/jthomp/file.diff"

gets the *file1* and *file2* files from the *convex* and *byu* machines, executes a *diff*(1) command and puts the results in *file.diff* in the local */usr/spool/uucppublic/jthomp/* directory.

Any special shell characters, such as <>;|, should be quoted either by quoting the entire *command-string* or quoting the special characters as individual arguments.

uux attempts to get all files to the execution system.

For example, assuming both the path and permissions are correct on systems a, b, and c and that they all have *uucp* connections between them, the command

uux "a!wc b!/usr/file1 > c!/usr/file2"

gets */usr/file1* from system *b* and sends it to system *a*, performs a *wc* command on that file, and sends the result of the *wc* command to system *c*.

uux notifies you by mail if the requested command on the remote system was disallowed. This notification can be turned off by the **-n** option.

The following *options* are interpreted by *uux*:

- The standard input to *uux* is made the standard input to the *command-string*.

-a*name*
Use *name* as the user identification, replacing the initiator user-id.

-c Do not copy local file to the spool directory for transfer to the remote machines. This is the default).

-C Force the copy of local files to the spool directory for transfer.

-g*grade*
grade is a single letter or number, from **0** to **9**, **A** to **Z**, or **a** to **z**; **0** is the highest, and **z** is the lowest grade. The default is **A**; by comparison *uucp*(1C) defaults to **n**, and mail is usually sent at grade **C**. Lower grades should be specified for high-volume jobs, such as news.

-l Try and make a link from the original file to the spool directory. If the link cannot be made, copy the file.

-n Do not notify the user when the command completes.

-p Same as -: The standard input to *uux* is made the standard input to the *command-*

string.

–r Do not start the file transfer, just queue the job.

–x*debug*
Produce debugging output on stdout. The debug is a number between 0 and 99; higher numbers give more detailed information. Debugging is permitted only for privileged users (specifically, those with read access to *L.sys*(5)).

–z Notify the user only if the command fails.

–L Start up *uucico* with the **-L** flag. This forces calls to be made to local sites only (see *uucico*(8C)).

FILES

/usr/spool/uucp	spool directories
/usr/lib/uucp/*	UUCP configuration data and daemons

SEE ALSO

uucp(1C), uucico(8C), uuxqt(8C).

WARNING

For security reasons, many installations limit the list of commands executable on behalf of an incoming request from *uux*. Many sites permit little more than the receipt of mail (see *mail*(1)) via *uux*.

BUGS

Only the first command of a shell pipeline may have a *system-name*!. All other commands are executed on the system of the first command.

The use of the shell metacharacter * will probably not do what you want it to do.

The shell tokens << and >> are not implemented.

When invoking *uux* from *csh*(1), the "!" character must be prefixed by the "\" escape to inhibit *csh*'s history mechanism. (Quotes are not sufficient.)

NAME

vers – set/display version numbers

SYNOPSIS

vers [**–v** *version*] *file* [*file* ...]

DESCRIPTION

Vers is used to set or display version numbers in object files, executable files and archives. Version numbers are of the form **X.X.X.X**, where **X** is an integer from 0 to 255.

When displayed, a version number will have at least the first 2 digits - the last 2 digits will be displayed if they are not zero. The **-v** switch sets the indicated file's version number to the specified value.

Note that if the **-v** switch is used on an archive file which has been run through *ranlib*, the archive file will have to be run through *ranlib* again. This is because *vers* changes the archive file, which causes the loader to think it is out of date.

In object and executable files, the version number is a reserved field in the file header (see *a.out*(5)).

In archives, version numbers are stored as text in a file named __.VERSION within the archive.

NOTE

Vers will not assign a version number to pre-SOFF object files.

SEE ALSO

a.out(5)

NAME

vi, view – screen oriented (visual) text editors based on ex

SYNOPSIS

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

view [vi options]

DESCRIPTION

Vi (visual) is a display oriented text editor based on *ex*(1). Since *ex* and *vi* run the same code, it is possible to get to the command mode of *ex* from within *vi* and vice-versa. The use of *vi* is described in the *ConvexOS Tutorial Papers.*

View forces a read-only mode of operation. That is, the files to be edited are treated as if they were write-protected. Otherwise, *view* performs the same functions as *vi.*

Options available with *vi* are:

–t	Edit the file containing the *tag* and position the editor at its definition.
–r	Recover file modifications after a system crash.
–R	Set *readonly* option which is used to read but not modify a file.
+command	Editor begins execution by processing the specified command.
–l	Sets the *showmatch* option.
-w*n*	Set the default window size to *n* .
–x	Editor will handle encrypted files. Note: this option is not available in the international distribution of ConvexOS.

FILES

See *ex*(1).

SEE ALSO

ex (1), edit (1), *vi Quick Reference*

AUTHOR

William Joy
Mark Horton added macros to *visual* mode.

BUGS

Software tabs using **^T** work only immediately after the *autoindent.*

Left and right shifts on intelligent terminals don't make use of insert and delete character operations in the terminal.

The *wrapmargin* option can be fooled since it looks at output columns when blanks are typed. If a long word passes through the margin and onto the next line without a break, then the line won't be broken.

Insert/delete within a line can be slow if tabs are present on intelligent terminals, since the terminals need help in doing this correctly.

Saving text on deletes in the buffers is inefficient.

Exiting *vi*/*ex* with **ZZ** or **:q** immediately after executing **:1,$d** results in no changes to the file, i.e. no lines of the file will be changed or modified in the editing session.

There is no way to use the **:append**, **:change**, and **:insert** commands, since it is not possible to give more than one line of input to a **:** escape. To use these on a **:global** you must **Q** to *ex* command mode, execute them, and then reenter the screen editor with *vi* or *open.*

Using "w" to jump over the last word on a line will not work if the line ends in a blank.

RESTRICTIONS

The "_" key (by itself) should not be mapped to anything else.

NAME

vmstat – report virtual memory statistics

SYNOPSIS

vmstat [[**–f**][**–s**][**–t**][**–z**]]
or
vmstat [**–S**][*interval* [*count*]]

DESCRIPTION

vmstat delves into the system and normally reports certain statistics kept about process, virtual memory, trap, and CPU activity.

If given a **–f** argument, it instead reports on the number of *forks* and *vforks* since system startup and the number of pages of virtual memory involved in each kind of fork. If given a **–s** argument, it instead prints the contents of the *cnt* structure, giving the total number of several kinds of paging-related events that have occurred since boot. If given a **–t** argument, it instead prints detailed statistic/tracing info kept only by appropriately instrumented kernels. The **–t** option works only on kernels with profiling turned on. If given a **–z** argument, it instead zeroes out the *_cnt* structure; or, if */vmunix* does not match the running system, some random locations in system memory (you must be the superuser, and be super cautious, for this option to have the desired effect).

Note that you cannot specify intervals if **-f**, **-s**, or **-z** are specified.

If none of these options are given, *vmstat* reports in the first line a summary of the virtual memory activity since the system has been booted. If *interval* is specified, successive lines are summaries over the last *interval* seconds. "vmstat 5" prints what the system is doing every five seconds; this is a good choice of printing interval because this is how often some of the statistics are sampled in the system; others vary every second, and running the output for a while will make it apparent which are recomputed every second. If a *count* is given, the statistics are repeated *count* times. The format fields are:

Procs: information about numbers of processes in various states.

r	in run queue
b	blocked for resources (I/O, paging, etc.)
w	runnable or short sleeper (< 20 secs) but partially swapped

Memory: information about the use of virtual and real memory.

avm	kbytes virtually allocated
fre	kbytes free

Page: information about page faults and paging activity, in units per second, averaged over the last *interval* seconds. The first interval is always seconds since boot.

re	pages reclaimed from the swap device queue
at	pages attached (not kept; always 0)
pi	pages paged in
po	pages paged out
fr	pages freed (not kept; always 0)
de	short term memory shortfall (not kept; always 0)
sr	resident pages scanned by pageout daemon

Faults: trap/interrupt rate in units per second, averaged over the last *interval* seconds. The first interval is always seconds since boot.

in	(non-clock) device interrupts
sy	system calls
cs	CPU context switches
vs	Vector register context switches

CPU: breakdown of percentage of use of CPU time over the last *interval* seconds. The first interval is always seconds since boot. On a multiprocessor system, the values will be averaged over all CPUs, so percentages will not be greater than 100.

us	user time for normal and nice'd processes
sy	system time
id	CPU idle

If the **–S** argument is specified, **vmstat** reports on swapping rather than paging activity. This option changes two fields in the "paging" display: rather than the "re" and "at" fields, **vmstat** reports "si" (swap-ins), and "so" (swap-outs).

FILES

/vmunix	system namelist
/dev/mem	kernel memory
/lib/kernsyms/symdata_*	kernel symbol addresses

NAME

w – who is on and what they are doing

SYNOPSIS

w [**–l**][**–u**] [**–h**] [**–s**] [user]

DESCRIPTION

W prints a summary of the current activity on the system, including what each user is doing. The heading line shows the current time of day, how long the system has been up, the number of users logged into the system, and the load averages. The load average numbers give the number of jobs in the run queue averaged over 1, 5 and 15 minutes.

The fields output are: the users login name, the name of the tty the user is on, the time of day the user logged on, the number of minutes since the user last typed anything, the CPU time used by all processes and their children on that terminal, the CPU time used by the currently active processes, the name and arguments of the current process.

The **–h** flag suppresses the heading. The **–s** flag asks for a short form of output. In the short form, the tty is abbreviated, the login time and cpu times are left off, as are the arguments to commands. **–l** gives the long output, which is the default. The **-u** flag produces the same output at *uptime*(1).

If a *user* name is included, the output will be restricted to that user.

FILES

/vmunix	system namelist
/dev/mem	kernel memory
/dev/swap	swap device
/etc/utmp	logged in user information
*/lib/kernsyms/symdata_**	kernel symbol addresses

SEE ALSO

who(1), finger(1), ps(1)

AUTHOR

Mark Horton

BUGS

The notion of the "current process" is muddy. The current algorithm is "the highest numbered process on the terminal that is not ignoring interrupts, or, if there is none, the highest numbered process on the terminal". This fails, for example, in critical sections of programs like the shell and editor, or when faulty programs running in the background fork and fail to ignore interrupts. (In cases where no process can be found, *w* prints "–".)

The CPU time is only an estimate, in particular, if someone leaves a background process running after logging out, the person currently on that terminal is "charged" with the time.

Background processes are not shown, even though they account for much of the load on the system.

Sometimes processes, typically those in the background, are printed with null or garbaged arguments. In these cases, the name of the command is printed in parentheses.

W does not know about the new conventions for detection of background jobs. It will sometimes find a background job instead of the right one.

NAME

wait – await completion of process

SYNOPSIS

wait

DESCRIPTION

Wait until all processes started with **&** have completed, and report on abnormal terminations.

Because the *wait*(2) system call must be executed in the parent process, the Shell itself executes *wait*, without creating a new process.

SEE ALSO

sh(1)

BUGS

Not all the processes of a 3- or more-stage pipeline are children of the Shell, and thus can't be waited for. (This bug does not apply to *csh*(1).)

NAME

wall – write to all users

SYNOPSIS

wall [file]

DESCRIPTION

Wall reads the contents of the specified file or standard input until an end-of-file. It then sends this message, preceded by 'Broadcast Message ...', to all logged in users. Non-printable characters will be deleted from the message before it is displayed to the users.

The sender should be superuser to override any protections the users may have invoked.

FILES

/dev/tty?
/etc/utmp

SEE ALSO

mesg(1), write(1)

DIAGNOSTICS

'Cannot send to ...' when the open on a user's tty file fails.

NAME

wc – word count

SYNOPSIS

wc [**–lwc**] [file ...]

DESCRIPTION

Wc counts lines, words and characters in the named files, or in the standard input if no name appears. A word is a maximal string of characters delimited by spaces, tabs or newlines.

If an argument beginning with one of "lwc" is present, the specified counts (lines, words, or characters) are selected by the letters **l**, **w**, or **c**. The default is **–lwc**. If more than one file is specified, the grand total of the individual file counts is also displayed.

NAME

what – show what versions of object modules were used to construct a file

SYNOPSIS

what file ...

DESCRIPTION

What reads each file and searches for sequences of the form "@(#)" as inserted by the source code control system. It then prints the remainder of the string after this marker, up to a null character, newline, double quote, or ">" character.

BUGS

As SCCS is not licensed with UNIX/32V, this is a rewrite of the *what* command which is part of SCCS, and may not behave exactly the same as that command does.

NAME

which – locate a program file including aliases and paths (csh only)

SYNOPSIS

which [name] ...

DESCRIPTION

Which takes a list of names and looks for the files which would be executed had these names been given as commands. Each argument is expanded if it is aliased, and searched for along the user's path. Both aliases and path are taken from the user's .cshrc file.

FILES

~/.cshrc source of aliases and path values

SEE ALSO

csh(1)

DIAGNOSTICS

A diagnostic is given for names which are aliased to more than a single word, or if an executable file with the argument name was not found in the path.

BUGS

Must be executed by a csh, since only csh's know about aliases. *Which* does not recognize the builtin functions of *csh*.

NOTES

Which is obsoleted by the *csh* builtin command *whence*.

NAME

who – who is on the system

SYNOPSIS

who [who-file] [**am i**]

DESCRIPTION

Who, without an argument, lists the login name, terminal name, and login time for each current user.

Without an argument, *who* examines the */etc/utmp* file to obtain its information. If a file is given, that file is examined. Typically the given file will be /usr/adm/wtmp, which contains a record of all the logins since it was created. Then *who* lists logins, logouts, and crashes since the creation of the wtmp file. Each login is listed with user name, terminal name (with '/dev/' suppressed), and date and time. When an argument is given, logouts produce a similar line without a user name. Reboots produce a line with 'x' in the place of the device name, and a fossil time indicative of when the system went down.

With two arguments, as in 'who am I' (and also 'who are you'), *who* tells who you are logged in as.

FILES

/etc/utmp

SEE ALSO

getuid(2), utmp(5)

NAME

whoami – print the effective or real current user ID

SYNOPSIS

whoami [**-n**] [**-u**]

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. If the **-n** flag is given, *whoami* prints the id in numeric form instead of string form. If the **-u** flag is given, the real uid is printed instead of the effective uid.

FILES

/etc/passwd Name data base

SEE ALSO

who(1)

NAME

whois – DARPA Internet user name directory service

SYNOPSIS

whois *name*

DESCRIPTION

The *whois* service displays information in the main NIC database. Each record in the database includes a handle (a unique identifier assigned to it by NIC), a user name, a record type, and information specific to the record type.

The following is an example of *whois* output:

```
UUNET Communications Services, Inc. (NET-UUNET-ETHER) UUNET-ETHER 192.48.96.0
UUNET Communications Services (UU-DOM)                          UU.NET
UUNET Communications (UUCP)      POSTMASTER@UUNET.UU.NET        (703) 876-5050
UUNET Communications (UUNET)     POSTMASTER@UUNET.UU.NET        (703) 876-5050
UUNET Technologies, Inc. (ALTER-DOM)                         ALTER.NET
UUNET (NET-UUNET-WAN)            UUNET-WAN                   137.39.0.0
```

To use *whois*, enter your target string. Unless directed otherwise with a keyword, *whois* searches for the target string in several fields—handle, name, nickname, host name, network address, etc.—and in all database records.

The search often finds more records than just the one you requested. To search for a single, specific record, put a "!" before your target, which should be the handle of the record you seek. In summary lines, the handle is shown in parenthesis following the user name. For example, entering **!ABC** requests the record whose handle is "ABC."

If you don't know exactly what you're looking for, you can do a "partial" search, i.e., you specify a partial target string, and *whois* searches for every item that starts with that string. Enter a partial string by following the target with a trailing dot (or dots). For example, entering **mack.** directs *whois* to find "mack," "mackall," "mackay," and so forth.

When *whois* finds a match, it displays the data in one of two ways: as a full, detailed display for a single match (with possible subdisplay), or as one- or two-line summaries for multiple matches.

KEYWORD OVERVIEW

Keywords fall into four categories: 1) those that specify a field to search, 2) those that specify a record type, 3) those that modify the interpretation of input or specify the type of output to produce, and 4) commands such as **help**, **quit**, and so forth.

Use the following three keywords (shown with their minimum abbreviations in all caps) to request that *whois* restrict its search to a certain field in the database:

HAndle or **!**	For example,	**!lynn**	or	**HA lynn**
NAme or leading **.**		**.lynn**	or	**NA lynn**
Mailbox or **@**		**lynn@host**		

To find only a certain type of record, use one of the following keywords:

Arpanet	**DOmain**	**GAteway**	**GRoup**
HOst	**IMp**	**MIlnet**	**NEtwork**
Organization	**PSn**	**TAc**	

See the section, "RECORD TYPES," for detailed descriptions of the different record types. See the section, "OUTPUT KEYWORDS," for a list of the input and output control keywords.

HANDLES

Each database record has a single field (its "handle") that uniquely identifies it. The handle is shown in parentheses following the record's name, as in:

```
Conway, Lynn (LYNN)          CONWAY@A.ISI.EDU          (313) 763-5509
              ^^^^^^^ this is the handle
```

For individual users, the handle is composed of the user's initials and a trailing number to make it unique, if necessary; for example, **JS473.**

Handles for other types of records look more like "NET-ARPANET" and "EDU-DOM," and though you can find these records by their handles, it is easier to use a record-type search; for example, **NET ARPANET** and **DOM EDU**. Sometimes a host record's handle comes from a host name no longer in use. Therefore, it is better to search for **host** *xxx* and use a current, legitimate name for the host instead of relying on the potentially stale information in the handle.

MAILBOXES

The mailbox field is routinely checked in a general search. If your target contains an "**@**," *whois* checks only the mailbox field. For example, if you entered the target, **joe**, *whois* would find mailboxes "joe@..." and "joe%...."

Mailboxes are stored in the database with the host part canonized, that is, made into an official host name. If you use a non-official host name or nickname, the name is automatically canonized for you before the search. If the named host is not known, *whois* performs the search on the exact target string anyway.

You may search for mailboxes in any of three ways:

user@ Search for mailboxes with a user name part of **user** on any host. This is a fast search.

@host Search for all mailboxes on the specified **host**. This is a slow search.

user@host Search for an exact mailbox match on both user and host parts. This method is fastest.

NAMES

Whether for individuals, machines, or organizations, most records contain a name field. For those that do not, *whois* displays "[No name]" in their record output. For individual users, the name is in last-first order, with any title coming at the end. For example,

```
Smith, John
Smith, John Q.
Smith, John Q., III
```

Other records typically have name fields such as

```
Smith College
Smith & James Co.
```

You may specify any part of a name up to a space or comma; it will match every name that begins with that target. For example, the target, **Smith**, matches all of the above records. If you entered the target, **Smith, John**, *whois* would find the three users above and plenty more. Additionally, you can use the target, **Smith, J**, to find all users named "Smith" who have a first initial of "J."

Partial searches can also be useful in locating records for a particular name. You will often find the record you're looking for by doing a partial search on the name field.

RECORD TYPES

Each database record contains a field that defines its record type. The main NIC database contains records for individual users, domains, groups, hosts, networks, organizations, PSNs (formerly called IMPs), TACs, and gateways. It is advantageous to specify which type of record you are looking for because it allows *whois* to present fewer and closer matches, and to perform extra searches on fields specific to that record type—fields not ordinarily searched in a general lookup.

The usual fields searched include the handle, user name, nickname, host name, network name, and network address. Exceptions are noted below.

To search for a specific type of record, prepend one of the following keywords (shown with their minimum abbreviations in all caps) to your target:

Arpanet *whois* searches host type records for hosts on network address 10, the ARPANET. See the description of host type records, below.

DOmain *whois* searches the usual fields in domain type records, as well as in the domain name field (**COM**, **MIT.EDU**, and so on) of other type records.

You can match on any portion of a domain that precedes the dot. For example, if you entered **DOM MIT**, *whois* would find the record for MIT.EDU.

Gateway *whois* searches the same fields in gateway type records as it does for host type records.

HOst *whois* searches host type records, as well as PSN, TAC, and gateway records. *whois* searches in the host name, nickname, network address, and domain name fields, matching text before the "." or "-" in the name. For example, if you entered **HO RUTGERS**, *whois* would find RUTGERS.EDU, RUTGERS-GW.RUTGERS.EDU, and RED.RUTGERS.EDU. The record for RED.RUTGERS.EDU matches because the name field for "RED" is "Rutgers University."

MIlnet *whois* searches host type records for hosts on network address 26, the MILNET.

NEtwork *whois* searches the network name field (ARPANET, SINET, and so forth) and the network address field (10.0.0.0, 128.15.0.0, and so forth). When you specify a network address, *whois* tries to match on the network number portion of the network address. For example, if you entered **net 128**, *whois* would find all networks with a first octet of 128.

Organization or Group

whois searches the usual fields in organization or group type records, as well as in the nickname field.

To obtain the entire membership list of a group or organization, precede the group or organization argument with an asterisk (*).

CAUTION: If there are a lot of members, displaying them takes a long time!

PSn *whois* searches for the target in the usual fields as well as in the nickname, PSN name, and PSN number fields. Some examples of PSN names are **SRI51** and **ROCK-ISLAND**. If you entered **psn 1**, *whois* would find OBERURSEL-IMP (MILNET) and UCLA-IMP (ARPANET).

Tac *whois* searches the same fields in TAC type records as it does for host type records, as well as in the TAC number field, which contains the BBN-assigned small number identifier for that TAC.

OUTPUT KEYWORDS

This last class of keywords controls how *whois* displays the results of a search. The keywords are:

EXPand or *

Directs *whois* to expand the long display for a single match to include all the subdisplays, without asking whether you wish to see them.

~ Directs *whois* not to show any subdisplays.

Full Directs *whois* to show a long display for every matching record, rather than only when there is a single match.

SUMmary Directs *whois* to show a summary line for each match, even if there is only one match.

SUBdisplay or **%**
Directs *whois* to show the subdisplay for the matching record following a summary line for the record itself.

SUBDISPLAYS

Some record types have associated subdisplays. *whois* usually asks you if you want subdisplays with a question such as "Would you like to see...?". If the **EXPand** or "*" keyword is set, *whois* shows the subdisplays without asking. The inverse keyword, "~," directs *whois* not to show subdisplays at all.

The following table lists record types that have associated subdisplays:

record type	subdisplay
host	registered users
network	hosts on network
top-level domain	sub-domains under domain
secondary domain	hosts in secondary domain
group/organization	registered members

You can use the keyword, **SUBdisplay** or "**%**," to show only a record's subdisplay, without showing the detailed display for the record itself. When used in conjunction with the keyword, **Full** or "=," the **SUBdisplay** keyword provides a way to display the full record for all users of a particular host, all members of an organization, and so forth, instead of having to enter all the handles or names individually.

NOTES

If you enter the command

whois help

whois displays information similar to that found in this man page.

SEE ALSO

RFC 954: Nicname/Whois

NAME

window – window environment

SYNOPSIS

window [**–t**] [**–f**] [**–d**] [**–e escape-char**] [**–c command**]

DESCRIPTION

window implements a window environment on ASCII terminals.

A window is a rectangular portion of the physical terminal screen associated with a set of processes. Its size and position can be changed by the user at any time. Processes communicate with their window in the same way they normally interact with a terminal — through their standard input, output, and diagnostic file descriptors. The window program handles the details of redirecting input an output to and from the windows. At any one time, only one window can receive input from the keyboard, but all windows can simultaneously send output to the display.

Windows can overlap and are framed as necessary. Each window is named by one of the digits "1" to "9". This one character identifier, as well as a user definable label string, are displayed with the window on the top edge of its frame. A window can be designated to be in the *foreground*, in which case it will always be on top of all normal, non-foreground windows, and can be covered only by other foreground windows. A window need not be completely within the edges of the terminal screen. Thus a large window (possibly larger than the screen) may be positioned to show only a portion of its full size.

Each window has a cursor and a set of control functions. Most intelligent terminal operations such as line and character deletion and insertion are supported. Display modes such as underlining and reverse video are available if they are supported by the terminal. In addition, similar to terminals with multiple pages of memory, each window has a text buffer which can have more lines than the window itself.

OPTIONS

When *window* starts up, the commands (see long commands below) contained in the file *.windowrc* in the user's home directory are executed. If it does not exist, two equal sized windows spanning the terminal screen are created by default.

The command line options are

–t Turn on terse mode (see *terse* command below).

–f Fast. Don't perform any startup action.

–d Ignore *.windowrc* and create the two default windows instead.

–e escape-char

Set the escape character to *escape-char*. *Escape-char* can be a single character, or in the form ^*X* where *X* is any character, meaning control-*X*.

–c command

Execute the string *command* as a long command (see below) before doing anything else.

PROCESS ENVIRONMENT

With each newly created window, a shell program is spawned with its process environment tailored to that window. Its standard input, output, and diagnostic file descriptors are bound to one end of either a pseudo-terminal (*pty* (4)) or a UNIX domain socket (*socketpair* (4)). If a pseudo-terminal is used, then its special characters and modes (see *stty* (1)) are copied from the physical terminal. A *termcap* (5) entry tailored to this window is created and passed as environment (*environ* (5)) variable *TERMCAP*. The termcap entry contains the window's size and characteristics as well as information from the physical terminal, such as the existence of underline, reverse video, and other display modes, and the codes produced by the terminal's function keys, if any. In addition, the window size attributes of the pseudo-terminal are set to reflect the size of this window, and updated whenever it is changed by the user. In particular, the editor *vi*

(1) uses this information to redraw its display. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

OPERATION

During normal execution, *window* can be in one of two states: conversation mode and command mode. In conversation mode, the terminal's real cursor is placed at the cursor position of a particular window — called the current window — and input from the keyboard is sent to the process in that window. The current window is always on top of all other windows, except those in foreground. In addition, it is set apart by highlighting its identifier and label in reverse video.

Typing *window*'s escape character (normally ^P) in conversation mode switches it into command mode. In command mode, the top line of the terminal screen becomes the command prompt window, and *window* interprets input from the keyboard as commands to manipulate windows.

There are two types of commands: short commands are usually one or two key strokes; long commands are strings either typed by the user in the command window (see the ":" command below), or read from a file (see *source* below).

SHORT COMMANDS

Below, # represents one of the digits "1" to "9" corresponding to the windows 1 to 9. ^*X* means control-*X*, where *X* is any character. In particular, ^^ is control-^. *Escape* is the escape key, or ^[.

Select window # as the current window and return to conversation mode.

%# Select window # but stay in command mode.

^^ Select the previous window and return to conversation mode. This is useful for toggling between two windows.

escape Return to conversation mode.

^P Return to conversation mode and write ^P to the current window. Thus, typing two ^P's in conversation mode sends one to the current window. If the *window* escape is changed to some other character, that character takes the place of ^P here.

? List a short summary of commands.

^L Redraw the screen.

q Exit *window*. Confirmation is requested.

^Z Suspend *window*.

w Create a new window. The user is prompted for the positions of the upper left and lower right corners of the window. The cursor is placed on the screen and the keys "h", "j", "k", and "l" move the cursor left, down, up, and right, respectively. The keys "H", "J", "K", and "L" move the cursor to the respective limits of the screen. Typing a number before the movement keys repeats the movement that number of times. Return enters the cursor position as the upper left corner of the window. The lower right corner is entered in the same manner. During this process, the placement of the new window is indicated by a rectangular box drawn on the screen, corresponding to where the new window will be framed. Typing escape at any point cancels this command.

This window becomes the current window, and is given the first available ID. The default buffer size is used (see *nline* command below).

Only fully visible windows can be created this way.

c# Close window #. The process in the window is sent the hangup signal (see *kill* (1)). *Csh* (1) should handle this signal correctly and cause no problems.

m# Move window # to another location. A box in the shape of the window is drawn on the screen to indicate the new position of the window, and the same keys as those for the *w*

command are used to position the box. The window can be moved partially off-screen.

M# Move window # to its previous position.

s# Change the size of window #. The user is prompted to enter the new lower right corner of the window. A box is drawn to indicate the new window size. The same keys used in *w* and *m* are used to enter the position.

S# Change window # to its previous size.

^Y Scroll the current window up by one line.

^E Scroll the current window down by one line.

^U Scroll the current window up by half the window size.

^D Scroll the current window down by half the window size.

^B Scroll the current window up by the full window size.

^F Scroll the current window down by the full window size.

h Move the cursor of the current window left by one column.

j Move the cursor of the current window down by one line.

k Move the cursor of the current window up by one line.

l Move the cursor of the current window right by one column.

^S Stop output in the current window.

^Q Start output in the current window.

: Enter a line to be executed as long commands. Normal line editing characters (erase character, erase word, erase line) are supported.

LONG COMMANDS

Long commands are a sequence of statements parsed much like a programming language, with a syntax similar to that of C. Numeric and string expressions and variables are supported, as well as conditional statements.

There are two data types: string and number. A string is a sequence of letters or digits beginning with a letter. "_" and "." are considered letters. Alternately, non-alphanumeric characters can be included in strings by quoting them in "" or escaping them with "\". In addition, the "\" sequences of C are supported, both inside and outside quotes (e.g., "\n" is a new line, "\r" a carriage return). For example, these are legal strings: abcde01234, "&#$^*&#", ab"$#"cd, ab\$\#cd, "/usr/ucb/window".

A number is an integer value in one of three forms: a decimal number, an octal number preceded by "0", or a hexadecimal number preceded by "0x" or "0X". The natural machine integer size is used (i.e., the signed integer type of the C compiler). As in C, a non-zero number represents a boolean true.

The character "#" begins a comment which terminates at the end of the line.

A statement is either a conditional or an expression. Expression statements are terminated with a new line or ";". To continue an expression on the next line, terminate the first line with "\".

CONDITIONAL STATEMENT

window has a single control structure: the fully bracketed if statement in the form

```
if <expr> then
        <statement>
        . . .
elsif <expr> then
        <statement>
        . . .
```

```
else
        <statement>
        . . .
endif
```

The *else* and *elsif* parts are optional, and the latter can be repeated any number of times. *<Expr>* must be numeric.

EXPRESSIONS

Expressions in *window* are similar to those in the C language, with most C operators supported on numeric operands. In addition, some are overloaded to operate on strings.

When an expression is used as a statement, its value is discarded after evaluation. Therefore, only expressions with side effects (assignments and function calls) are useful as statements.

Single valued (no arrays) variables are supported, of both numeric and string values. Some variables are predefined. They are listed below.

The operators in order of increasing precedence:

<expr1> = <expr2>
Assignment. The variable of name *<expr1>*, which must be string valued, is assigned the result of *<expr2>*. Returns the value of *<expr2>*.

<expr1> ? <expr2> : <expr3>
Returns the value of *<expr2>* if *<expr1>* evaluates true (non-zero numeric value); returns the value of *<expr3>* otherwise. Only one of *<expr2>* and *<expr3>* is evaluated. *<Expr1>* must be numeric.

<expr1> || <expr2>
Logical or. Numeric values only. Short circuit evaluation is supported (i.e., if *<expr1>* evaluates true, then *<expr2>* is not evaluated).

<expr1> && <expr2>
Logical and with short circuit evaluation. Numeric values only.

<expr1> | <expr2>
Bitwise or. Numeric values only.

<expr1> ^ <expr2>
Bitwise exclusive or. Numeric values only.

<expr1> & <expr2>
Bitwise and. Numeric values only.

<expr1> == <expr2>, <expr1> != <expr2>
Comparison (equal and not equal, respectively). The boolean result (either 1 or 0) of the comparison is returned. The operands can be numeric or string valued. One string operand forces the other to be converted to a string in necessary.

<expr1> < <expr2>, <expr1> > <expr2>, <expr1> <= <expr2>,
Less than, greater than, less than or equal to, greater than or equal to. Both numeric and string values, with automatic conversion as above.

<expr1> << <expr2>, <expr1> >> <expr2>
If both operands are numbers, *<expr1>* is bit shifted left (or right) by *<expr2>* bits. If *<expr1>* is a string, then its first (or last) *<expr2>* characters are returns (if *<expr2>* is also a string, then its length is used in place of its value).

<expr1> + <expr2>, <expr1> - <expr2>
Addition and subtraction on numbers. For "+", if one argument is a string, then the other is converted to a string, and the result is the concatenation of the two strings.

<expr1> * <expr2>, <expr1> / <expr2>, <expr1> % <expr2>

Multiplication, division, modulo. Numbers only.

-<expr>, ~<expr>, !<expr>, $<expr>, $?<expr>
The first three are unary minus, bitwise complement and logical complement on numbers only. The operator, "$", takes *<expr>* and returns the value of the variable of that name. If *<expr>* is numeric with value *n* and it appears within an alias macro (see below), then it refers to the nth argument of the alias invocation. "$?" tests for the existence of the variable *<expr>*, and returns 1 if it exists or 0 otherwise.

<expr>(<arglist>)
Function call. *<Expr>* must be a string that is the unique prefix of the name of a builtin *window* function or the full name of a user defined alias macro. In the case of a builtin function, *<arglist>* can be in one of two forms:

<expr1>, <expr2>, . . .
argname1 = <expr1>, argname2 = <expr2>, . . .

The two forms can in fact be intermixed, but the result is unpredictable. Most arguments can be omitted; default values will be supplied for them. The *argnames* can be unique prefixes of the argument names. The commas separating arguments are used only to disambiguate, and can usually be omitted.

Only the first argument form is valid for user defined aliases. Aliases are defined using the *alias* builtin function (see below). Arguments are accessed via a variant of the variable mechanism (see "$" operator above).

Most functions return value, but some are used for side effect only and so must be used as statements. When a function or an alias is used as a statement, the parenthesis surrounding the argument list may be omitted. Aliases return no value.

BUILTIN FUNCTIONS

The arguments are listed by name in their natural order. Optional arguments are in square brackets ("[ ]"). Arguments that have no names are in angle brackets ("<>").

alias([<string>], [<string-list>])
If no argument is given, all currently defined alias macros are listed. Otherwise, *<string>* is defined as an alias, with expansion *<string-list>*. The previous definition of *<string>*, if any, is returned. Default for *<string-list>* is no change.

close(<window-list>)
Close the windows specified in *<window-list>*. If *<window-list>* is the word *all*, than all windows are closed. No value is returned.

cursormodes([modes])
Set the window cursor to *modes*. *Modes* is the bitwise or of the mode bits defined as the variables *m_ul* (underline), *m_rev* (reverse video), *m_blk* (blinking), and *m_grp* (graphics, terminal dependent). Return value is the previous modes. Default is no change. For example, cursor($m_rev|$m_blk) sets the window cursors to blinking reverse video.

echo([window], [<string-list>])
Write the list of strings, *<string-list>*, to *window*, separated by spaces and terminated with a new line. The strings are only displayed in the window, the processes in the window are not involved (see *write* below). No value is returned. Default is the current window.

escape([escapec])
Set the escape character to *escape-char*. Returns the old escape character as a one character string. Default is no change. *Escapec* can be a string of a single character, or in the form ^*X*, meaning control-*X*.

foreground([window], [flag])
Move *window* in or out of foreground. *Flag* can be one of *on*, *off*, *yes*, *no*, *true*, or *false*,

with obvious meanings, or it can be a numeric expression, in which case a non-zero value is true. Returns the old foreground flag as a number. Default for *window* is the current window, default for *flag* is no change.

label([window], [label])
Set the label of *window* to *label*. Returns the old label as a string. Default for *window* is the current window, default for *label* is no change. To turn off a label, set it to an empty string ("").

list() No arguments. List the identifiers and labels of all windows. No value is returned.

nline([nline])
Set the default buffer size to *nline*. Initially, it is 48 lines. Returns the old default buffer size. Default is no change. Using a very large buffer can slow the program down considerably.

select([window])
Make *window* the current window. The previous current window is returned. Default is no change.

shell([<string-list>])
Set the default window shell program to *<string-list>*. Returns the first string in the old shell setting. Default is no change. Initially, the default shell is taken from the environment variable *SHELL*.

source(filename)
Read and execute the long commands in *filename*. Returns -1 if the file cannot be read, 0 otherwise.

terse([flag])
Set terse mode to *flag*. In terse mode, the command window stays hidden even in command mode, and errors are reported by sounding the terminal's bell. *Flag* can take on the same values as in *foreground* above. Returns the old terse flag. Default is no change.

unalias(alias)
Undefine *alias*. Returns -1 if *alias* does not exist, 0 otherwise.

unset(variable)
Undefine *variable*. Returns -1 if *variable* does not exist, 0 otherwise.

variables()
No arguments. List all variables. No value is returned.

window([row], [column], [nrow], [ncol], [nline], [frame], [pty], [mapnl], [shell])
Open a window with upper left corner at *row, column* and size *nrow, ncol*. If *nline* is specified, then that many lines are allocated for the text buffer. Otherwise, the default buffer size is used. Default values for *row, column, nrow,* and *ncol* are, respectively, the upper, left-most, lower, or right-most extremes of the screen. *Frame, pty,* and *mapnl* are flag values interpreted in the same way as the argument to *foreground* (see above); they mean, respectively, put a frame around this window (default true), allocate pseudo-terminal for this window rather than socketpair (default true), and map new line characters in this window to carriage return and line feed (default true if socketpair is used, false otherwise). *Shell* is a list of strings that will be used as the shell program to place in the window (default is the program specified by *shell*, see below). The created window's identifier is returned as a number.

write([window], [<string-list>])
Send the list of strings, *<string-list>*, to *window*, separated by spaces but not terminated with a new line. The strings are actually given to the window as input. No value is returned. Default is the current window.

PREDEFINED VARIABLES

These variables are for information only. Redefining them does not affect the internal operation of *window*.

baud The baud rate as a number between 50 and 38400.

modes The display modes (reverse video, underline, blinking, graphics) supported by the physical terminal. The value of *modes* is the bitwise or of some of the one bit values, *m_blk*, *m_grp*, *m_rev*, and *m_ul* (see below). These values are useful in setting the window cursors' modes (see *cursormodes* above).

m_blk The blinking mode bit.

m_grp The graphics mode bit (not very useful).

m_rev The reverse video mode bit.

m_ul The underline mode bit.

ncol The number of columns on the physical screen.

nrow The number of rows on the physical screen.

term The terminal type. The standard name, found in the second name field of the terminal's *TERMCAP* entry, is used.

FILES

~/.windowrc	startup command file.
/usr/lib/windowrc	sample startup command file.
/dev/[pt]ty[pq]?	pseudo-terminal devices.

RESTRICTIONS

Remote login shells do not inherit the terminal information.

DIAGNOSTICS

Should be self explanatory.

NAME

write – write to another user

SYNOPSIS

write user [ttyname]

DESCRIPTION

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

Message from yoursystem!yourname yourttyname...

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 'EOF' on the other terminal and exits.

If you want to write to a user who is logged in more than once, the *ttyname* argument may be used to indicate the appropriate terminal name. Terminal names are specified by *ttyxxx* where *xxx* is the *tty* number.

Permission to write may be denied or granted by use of the *mesg* command. Certain commands, in particular *nroff* and *pr*(1) disallow messages in order to prevent messy output.

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 him to write back before starting to send. Each party should end each message with a distinctive signal—**(o)** for 'over' is conventional—that the other may reply. **(oo)** for 'over and out' is suggested when conversation is about to be terminated.

FILES

/etc/utmp	to find user
/bin/sh	to execute '!'

SEE ALSO

mesg(1), who(1), mail(1)

NAME

xstr – extract strings from C programs to implement shared strings

SYNOPSIS

xstr [**–c**] [–] [file]

DESCRIPTION

Xstr maintains a file *strings* into which strings in component parts of a large program are hashed. These strings are replaced with references to this common area. This serves to implement shared constant strings, most useful if they are also read-only.

The command

xstr –c name

will extract the strings from the C source in name, replacing string references by expressions of the form (&xstr[number]) for some number. An appropriate declaration of *xstr* is prepended to the file. The resulting C text is placed in the file *x.c*, to then be compiled. The strings from this file are placed in the *strings* data base if they are not there already. Repeated strings and strings which are suffices of existing strings do not cause changes to the data base.

After all components of a large program have been compiled a file *xs.c* declaring the common *xstr* space can be created by a command of the form

xstr

This *xs.c* file should then be compiled and loaded with the rest of the program. If possible, the array can be made read-only (shared) saving space and swap overhead.

Xstr can also be used on a single file. A command

xstr name

creates files *x.c* and *xs.c* as before, without using or affecting any *strings* file in the same directory.

It may be useful to run *xstr* after the C preprocessor if any macro definitions yield strings or if there is conditional code which contains strings which may not, in fact, be needed. *Xstr* reads from its standard input when the argument '–' is given. An appropriate command sequence for running *xstr* after the C preprocessor is:

```
cc –E name.c | xstr –c –
cc –c x.c
mv x.o name.o
```

Xstr does not touch the file *strings* unless new items are added, thus *make* can avoid remaking *xs.o* unless truly necessary.

FILES

strings	Data base of strings
x.c	Massaged C source
xs.c	C source for definition of array 'xstr'
/tmp/xs*	Temp file when 'xstr name' doesn't touch *strings*

SEE ALSO

mkstr(1)

AUTHOR

William Joy

BUGS

If a string is a suffix of another string in the data base, but the shorter string is seen first by *xstr* both strings will be placed in the data base, when just placing the longer one there will do.

NAME

yacc – yet another compiler-compiler

SYNOPSIS

yacc [**–vd**] 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.

FILES

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

SEE ALSO

lex(1)
LR Parsing by A. V. Aho and S. C. Johnson, Computing Surveys, June, 1974.
"YACC – Yet Another Compiler Compiler" in the *ConvexOS Tutorial Papers*

DIAGNOSTICS

The number of reduce-reduce and shift-reduce conflicts is reported on the standard 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

SYNOPSIS

yes [*expletive*]

DESCRIPTION

Yes repeatedly outputs "y", or if *expletive* is given, that is output repeatedly. Termination is by rubout.

NAME

yesterday – show yesterday's date

SYNOPSIS

yesterday

DESCRIPTION

Yesterday's date is printed to stdout.
Format: Weekday Month Day Year (ie. Tue Oct 21 1986)

SEE ALSO

date(1)

Section 7
Text processing/ terminal environments

NAME

miscellaneous – miscellaneous useful information pages

DESCRIPTION

This section contains miscellaneous documentation, mostly in the area of text processing macro packages for *nroff*(1).

ascii	map of ASCII character set
environ	user environment
eqnchar	special character definitions for eqn
hier	file system hierarchy
mailaddr	mail addressing description
man	macros to typeset manual pages
me	macros for formatting papers
ms	macros for formatting manuscripts
term	conventional names for terminals

NAME

ascii – map of ASCII character set

DESCRIPTION

ascii is a map of the ASCII character set, to be printed as needed. It contains:

```
|000 nul|001 soh|002 stx|003 etx|004 eot|005 enq|006 ack|007 bel|
|010 bs |011 ht |012 nl |013 vt |014 np |015 cr |016 so |017 si |
|020 dle|021 dc1|022 dc2|023 dc3|024 dc4|025 nak|026 syn|027 etb|
|030 can|031 em |032 sub|033 esc|034 fs |035 gs |036 rs |037 us |
|040 sp |041  ! |042  " |043  # |044  $ |045  % |046  & |047  ´ |
|050  ( |051  ) |052  * |053  + |054  , |055  - |056  . |057  / |
|060  0 |061  1 |062  2 |063  3 |064  4 |065  5 |066  6 |067  7 |
|070  8 |071  9 |072  : |073  ; |074  < |075  = |076  > |077  ? |
|100  @ |101  A |102  B |103  C |104  D |105  E |106  F |107  G |
|110  H |111  I |112  J |113  K |114  L |115  M |116  N |117  O |
|120  P |121  Q |122  R |123  S |124  T |125  U |126  V |127  W |
|130  X |131  Y |132  Z |133  [ |134  \ |135  ] |136  ^ |137  _ |
|140  ` |141  a |142  b |143  c |144  d |145  e |146  f |147  g |
|150  h |151  i |152  j |153  k |154  l |155  m |156  n |157  o |
|160  p |161  q |162  r |163  s |164  t |165  u |166  v |167  w |
|170  x |171  y |172  z |173  { |174  | |175  } |176  ~ |177 del|

| 00 nul| 01 soh| 02 stx| 03 etx| 04 eot| 05 enq| 06 ack| 07 bel|
| 08 bs | 09 ht | 0a nl | 0b vt | 0c np | 0d cr | 0e so | 0f si |
| 10 dle| 11 dc1| 12 dc2| 13 dc3| 14 dc4| 15 nak| 16 syn| 17 etb|
| 18 can| 19 em | 1a sub| 1b esc| 1c fs | 1d gs | 1e rs | 1f us |
| 20 sp | 21  ! | 22  " | 23  # | 24  $ | 25  % | 26  & | 27  ´ |
| 28  ( | 29  ) | 2a  * | 2b  + | 2c  , | 2d  - | 2e  . | 2f  / |
| 30  0 | 31  1 | 32  2 | 33  3 | 34  4 | 35  5 | 36  6 | 37  7 |
| 38  8 | 39  9 | 3a  : | 3b  ; | 3c  < | 3d  = | 3e  > | 3f  ? |
| 40  @ | 41  A | 42  B | 43  C | 44  D | 45  E | 46  F | 47  G |
| 48  H | 49  I | 4a  J | 4b  K | 4c  L | 4d  M | 4e  N | 4f  O |
| 50  P | 51  Q | 52  R | 53  S | 54  T | 55  U | 56  V | 57  W |
| 58  X | 59  Y | 5a  Z | 5b  [ | 5c  \ | 5d  ] | 5e  ^ | 5f  _ |
| 60  ` | 61  a | 62  b | 63  c | 64  d | 65  e | 66  f | 67  g |
| 68  h | 69  i | 6a  j | 6b  k | 6c  l | 6d  m | 6e  n | 6f  o |
| 70  p | 71  q | 72  r | 73  s | 74  t | 75  u | 76  v | 77  w |
| 78  x | 79  y | 7a  z | 7b  { | 7c  | | 7d  } | 7e  ~ | 7f del|
```

FILES

/usr/man/man7/ascii.7

NAME

environ – user environment

SYNOPSIS

extern char **environ;

DESCRIPTION

An array of strings called the "environment" is made available by *execve*(2) when a process begins. By convention these strings have the form *name* = *value*. The following names are used by various commands:

PATH The sequence of directory prefixes that *sh*, *time*, *nice*(1), etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by a colon (:). *login*(1) sets *PATH=:/usr/ucb:/bin:/usr/bin*.

HOME A user's login directory, set by *login*(1) from the password file *passwd*(5).

TERM The kind of terminal for which output is to be prepared. This information is used by commands, such as *nroff*, which may exploit special terminal capabilities. See */etc/termcap* (*termcap*(5)) for a list of terminal types.

SHELL The file name of the users login shell.

TERMCAP

The string describing the terminal in TERM, or the name of the termcap file, see *termcap*(5),*termcap*(3X).

EXINIT A startup list of commands read by *ex*(1), *edit*(1), and *vi*(1).

USER The login name of the user.

PRINTER The name of the default printer to be used by *lpr*(1), *lpq*(1), and *lprm*(1).

Further names may be placed in the environment by the *export* command and *name=value* arguments in *sh*(1), or by the *setenv* command if you use *csh*(1). Arguments may also be placed in the environment at the point of an *execve*(2). It is unwise to conflict with certain *sh*(1) variables that are frequently exported by *.profile* files: MAIL, PS1, PS2, IFS.

SEE ALSO

csh(1), ex(1), login(1), sh(1), execve(2), system(3), termcap(3X), termcap(5)

NAME

eqnchar – special character definitions for eqn

SYNOPSIS

eqn /usr/pub/eqnchar [files] | **troff** [options]

neqn /usr/pub/eqnchar [files] | **nroff** [options]

DESCRIPTION

eqnchar contains *troff* and *nroff* character definitions for constructing characters that are not available on the Graphic Systems typesetter. These definitions are primarily intended for use with *eqn* and *neqn*. It contains definitions for the following characters

ciplus	⊕	*||*	‖	*square*	□
citimes	⊗	*langle*	⟨	*circle*	○
wig	∼	*rangle*	⟩	*blot*	▫
-wig	≃	*hbar*	ħ	*bullet*	•
>wig	≳	*ppd*	⊥	*prop*	∝
<wig	≲	*<->*	↔	*empty*	∅
=wig	≅	*<=>*	⇔	*member*	∈
star	✱	*|<*	⊀	*nomem*	∉
bigstar	✳	*|>*	⊁	*cup*	∪
=dot	≐	*ang*	∠	*cap*	∩
orsign	∨	*rang*	∟	*incl*	⊆
andsign	∧	*3dot*	⋮	*subset*	⊂
=del	≜	*thf*	∴	*supset*	⊃
oppA	∀	*quarter*	¼	*!subset*	⊄
oppE	∃	*3quarter*	¾	*!supset*	⊅
angstrom	Å	*degree*	°		

FILES

/usr/pub/eqnchar

SEE ALSO

troff(1), eqn(1) (optional products)

NAME

hier -- file system hierarchy

DESCRIPTION

The following outline gives a quick tour through a representative directory hierarchy:

/ root
/vmunix
the kernel binary (ConvexOS itself)
/lost+found
directory for connecting detached files for
fsck(8)
/dev/ devices (4)
MAKEDEV
shell script to create special files
MAKEDEV.local
site specific part of MAKEDEV
console main console,
tty(4)
tty* terminals,
tty(4)
...
/bin/ utility programs, cf */usr/bin/* (1)
as assembler
cc C compiler executive, cf */lib/ccom*, */lib/cpp*, */lib/c2*
csh C shell
...
/lib/ object libraries and other stuff, cf /usr/lib/
libc.a system calls, standard I/O, etc. (2,3,3S)
...
ccom C compiler proper
cpp C preprocessor
...
/etc/ essential data and maintenance utilities; sect (8), cf /usr/etc/
dump dump program
dump(8)
passwd password file,
passwd(5)
group group file,
group(5)
motd message of the day,
login(1)
termcap description of terminal capabilities,
termcap(5)
ttytype table of what kind of terminal is on each port,
ttytype(5)
mtab mounted file table,
mtab(5)
dumpdates
dump history,
dump(8)
fstab file system configuration table
fstab(5)
disktab disk characteristics and partition tables,
disktab(5)

hosts host name to network address mapping file,
hosts(5)
networks
network name to network number mapping file,
networks(5)
protocols
protocol name to protocol number mapping file,
protocols(5)
services network services definition file,
services(5)
remote names and description of remote hosts for
tip(1C),
remote(5)
phones private phone numbers for remote hosts, as described in
phones(5)
ttys properties of terminals,
ttys(5)
getty part of
login,
getty(8)
init the parent of all processes,
init(8)
rc shell program to bring the system up
rc.local site dependent portion of
rc
cron the clock daemon,
cron(8)
mount *mount*(8)
...
/sys/ system build directory
/tmp/ temporary files, usually on a fast device, cf */usr/tmp/*
e* used by
ed(1)
ctm* used by
cc(1)
...
/usr/ general-pupose directory, usually a mounted file system
adm/ administrative information
wtmp login history,
utmp(5)
messages
hardware error messages
bin/
utility programs, to keep */bin/ small*
tmp/ temporaries, to keep */tmp/ small*
convex/ CONVEX specific utilities
ansitar ansi labeled tape utility
bprof source level profiler
...
dict/ word lists, etc.
words principal word list, used by
look(1)
spellhist

history file for
spell(1)
include/ standard #include files
a.out.h object file layout,
a.out(5)
stdio.h standard I/O,
intro(3S)
math.h (3M)
...
sys/ system-defined layouts
machine/
machine-defined layouts
...
lib/ object libraries and files, to keep */lib/ small*
atrun scheduler for
at(1)
lint/ utility files for lint
lint[12] subprocesses for
lint(1)
llib-lc dummy declarations for */lib/libc.a*, used by
lint(1)
llib-lm dummy declarations for */lib/libc.m*
...
tmac/ macros for
nroff(1)
tmac.an macros for
man(7)
tmac.s macros for
ms(7)
...
uucp/ programs and data for
uucp(1C)
L.sys remote system names and numbers
uucico the real copy program
...
units conversion tables for
units(1)
skel/ skeleton initialization files copied by *nu(8)*.

etc/ more system files and programs, to keep */etc* small
dumpfs dumps filesystem information
quota programs to manipulate the quota system
in.* internet daemons started up by *inetd*(8C)
magic database of magic numbers used by *file*(1)
queued the batch system queue daemon
rpc.* RPC daemons started up by *inetd*(8C)
spd the batch system service point daemon
...
/usr/ man/
volume 1 of this manual,
man(1)
whatis default whatis database, text version
whatis.dir

```
            whatis.pag
                    default whatis database, dbm version
            man.template
                    template for manual page
            man1/   chapter 1
                    as.1
                    mount.1m
                    ...
            ...
            manl/   local manual pages
            cat1/   preformatted pages for section 1
            ...
            idx1/   indices for section headers for section 1
            ...
        msgs/   messages, cf
            msgs(1)
            bounds  highest and lowest message
        preserve/
            editor temporaries preserved here after crashes/hangups
        spool/  delayed execution files
            at/     used by
                    at(1)
            lpd/    used by
                    lpr(1)
                    lock    present when line printer is active
                    cf*     copy of file to be printed, if necessary
                    df*     daemon control file,
                            lpd(8)
                    tf*     transient control file, while
                            lpr
                            is working
            uucp/   work files and staging area for
                    uucp(1C)
                    LOGFILE
                            summary log
                    LOG.*   log file for one transaction
            mail/   mailboxes for
                    mail(1)
                    name
                            mail file for user
                            name
                    name.lock
                            lock file while
                            name
                            is receiving mail
                    mqueue/
                            mail queue for
                            sendmail(8)
            ...
    ucb/    binaries of programs developed at UCB
        ...
        edit    editor for beginners
        ex      command editor for experienced users
```

...
mail mail reading/sending subsystem
man on line documentation
...
vi visual editor
/mnt/
mounted file system which contains user directories
wd initial working directory of a user,
typically
wd
is the user's login name
.profile set environment for
sh(1),
environ(7)
.project what you are doing (used by
finger(1)
)
.cshrc startup file for
csh(1)
.exrc startup file for
ex(1)
.plan what your short-term plans are (used by
finger(1)
)
.netrc startup file for various network programs
.msgsrc startup file for
msgs(1)
.mailrc startup file for
mail(1)
calendar
user's datebook for
calendar(1)

SEE ALSO

ls(1), apropos(1), whatis(1), finger(1), which(1), find(1), grep(1)

BUGS

The position of files is subject to change without notice.

NAME

hostname – host name resolution description

DESCRIPTION

Hostnames are domains, where a domain is a hierarchical, dot-separated list of subdomains; for example, the machine monet, in the Berkeley subdomain of the EDU subdomain of the ARPANET would be represented as

monet.Berkeley.EDU

(with no trailing dot).

Hostnames are often used with network client and server programs, which must generally translate the name to an address for use. (This function is generally performed by the library routine *gethostbyname* (3).) Hostnames are resolved by the internet name resolver in the following fashion.

If the name consists of a single component, i.e. contains no dot, and if the environment variable "HOSTALIASES" is set to the name of a file, that file is searched for an string matching the input hostname. The file should consist of lines made up of two white-space separated strings, the first of which is the hostname alias, and the second of which is the complete hostname to be substituted for that alias. If a case-sensitive match is found between the hostname to be resolved and the first field of a line in the file, the substituted name is looked up with no further processing.

If the input name ends with a trailing dot, the trailing dot is removed, and the remaining name is looked up with no further processing.

If the input name does not end with a trailing dot, it is looked up in the local domain and its parent domains until either a match is found or fewer than 2 components of the local domain remain. For example, in the domain CS.Berkeley.EDU, the name lithium.CChem will be checked first as lithium.CChem.CS.Berkeley.EDU and then as lithium.CChem.Berkeley.EDU. Lithium.CChem.EDU will not be tried, as the there is only one component remaining from the local domain.

SEE ALSO

gethostbyname(3), resolver(5), mailaddr(7), named(8), RFC883

NAME

mailaddr – mail addressing description

DESCRIPTION

Mail addresses are based on the ARPANET protocol listed at the end of this manual page. These addresses are in the general format

user@domain

where a domain is a hierarchical dot separated list of subdomains. For example, the address

eric@monet.berkeley.edu

is normally interpreted from right to left: the message should go to the ARPA name tables (which do not correspond exactly to the physical ARPANET), then to the Berkeley gateway, after which it should go to the local host monet. When the message reaches monet it is delivered to the user "eric".

Unlike some other forms of addressing, this does not imply any routing. Thus, although this address is specified as an ARPA address, it might travel by an alternate route if that were more convenient or efficient. For example, at Berkeley, the associated message would probably go directly to monet over the Ethernet rather than going via the Berkeley ARPANET gateway.

Abbreviation.

Under certain circumstances it may not be necessary to type the entire domain name. In general, anything following the first dot may be omitted if it is the same as the domain from which you are sending the message. For example, a user on "calder.berkeley.edu" could send to "eric@monet" without adding the "berkeley.edu" since it is the same on both sending and receiving hosts.

Certain other abbreviations may be permitted as special cases. For example, at Berkeley, ARPANET hosts may be referenced without adding the "berkeley.edu" as long as their names do not conflict with a local host name.

Compatibility.

Certain old address formats are converted to the new format to provide compatibility with the previous mail system. In particular,

user@host.ARPA

is allowed and

host:user

is converted to

user@host

to be consistent with the *rcp*(1) command.

Also, the syntax

host!user

is converted to:

user@host.UUCP

This is normally converted back to the "host!user" form before being sent on for compatibility with older UUCP hosts.

The current implementation is not able to route messages automatically through the UUCP network. Until that time you must explicitly tell the mail system which hosts to send your message through to get to your final destination.

Case Distinctions.

Domain names (i.e., anything after the "@" sign) may be given in any mixture of upper and lower case with the exception of sites in the .UUCP pseudo domain. User names are required to be case sensitive, although some hosts may relax this requirement.

Route-addrs.

Under some circumstances it may be necessary to route a message through several hosts to get it to the final destination. Normally this routing is done automatically, but sometimes it is desirable to route the message manually. Addresses which show these relays are termed "route-addrs." These use the syntax:

<@hosta,@hostb:user@hostc>

This specifies that the message should be sent to hosta, from there to hostb, and finally to hostc. This path is forced even if there is a more efficient path to hostc.

Route-addrs occur frequently on return addresses, since these are generally augmented by the software at each host. It is generally possible to ignore all but the "user@domain" part of the address to determine the actual sender.

Route-addrs in this form have been declared obsolescent by RFC822, in appendix C, section 5.4. Another solution to the route address problem is to use an address in the form of user%siteb@sitea.domain, although this to is considered obsolescent.

Postmaster.

Every site is required to have a user or user alias designated "postmaster" to which problems with the mail system may be addressed.

Other Networks.

Some other networks can be reached by giving the name of the network as the last component of the domain. *This is not a standard feature* and may not be supported at all sites. For example, messages to CSNET or BITNET sites can often be sent to "user@host.CSNET" or "user@host.BITNET" respectively.

BUGS

The RFC822 group syntax ("group:user1,user2,user3;") is not supported except in the special case of "group:;" because of a conflict with old berknet-style addresses.

Route-Address syntax is grotty and obsolete.

UUCP- and ARPANET-style addresses do not coexist politely.

SEE ALSO

mail(1), sendmail(8); Crocker, D. H., *Standard for the Format of Arpa Internet Text Messages*, RFC822.

NAME

man – macros to typeset manual

SYNOPSIS

nroff –man file ...

DESCRIPTION

These macros are used to lay out pages of this manual. To see how this man page was constructed using these macros, see file */usr/man/man7/man.7*.

Any text argument *t* may be zero to six words. Quotes may be used to include blanks in a "word". If *text* is empty, the special treatment is applied to the next input line with text to be printed. In this way .I may be used to italicize a whole line, or .SM followed by .B to make small bold letters.

A prevailing indent distance is remembered between successive indented paragraphs, and is reset to default value upon reaching a nonindented paragraph. Default units for indents *i* are en's.

Type font and size are reset to default values before each paragraph, and after processing font and size setting macros.

These strings are predefined by **–man**:

*R '®', '(Reg)' in *nroff*.

*S Change to default type size.

FILES

/usr/lib/tmac/tmac.an
/usr/man/man7/man.7

SEE ALSO

man(1)

BUGS

Relative indents don't nest.

REQUESTS

Request	Cause Break	If no Argument	Explanation
.B *t*	no	*t*=n.t.l.*	Text *t* is bold.
.BI *t*	no	*t*=n.t.l.	Join words of *t* alternating bold and italic.
.BR *t*	no	*t*=n.t.l.	Join words of *t* alternating bold and Roman.
.DT	no	.5i 1i...	Restore default tabs.
.HP *i*	yes	*i*=p.i.*	Set prevailing indent to *i*. Begin paragraph with hanging indent.
.I *t*	no	*t*=n.t.l.	Text *t* is italic.
.IB *t*	no	*t*=n.t.l.	Join words of *t* alternating italic and bold.
.IP *x i*	yes	*x*=""	Same as .TP with tag *x*.
.IR *t*	no	*t*=n.t.l.	Join words of *t* alternating italic and Roman.
.LP	yes	-	Same as .PP.
.PD *d*	no	*d*=.4v	Interparagraph distance is *d*.
.PP	yes	-	Begin paragraph. Set prevailing indent to .5i.
.RE	yes	-	End of relative indent. Set prevailing indent to amount of starting .RS.
.RB *t*	no	*t*=n.t.l.	Join words of *t* alternating Roman and bold.
.RI *t*	no	*t*=n.t.l.	Join words of *t* alternating Roman and italic.
.RS *i*	yes	*i*=p.i.	Start relative indent, move left margin in distance *i*. Set prevailing indent to .5i for nested indents.
.SH *t*	yes	*t*=n.t.l.	Subhead.
.SM *t*	no	*t*=n.t.l.	Text
.SS *t*	no	-	Subsection; text *t* in bold.
.TH *n c v m*	yes	-	Begin page named *n* of chapter *c; x* is extra commentary, e.g. 'local', for page foot center; *v* alters page foot left, e.g. '4th Berkeley Distribution'; *m* alters page head center, e.g.

			'Brand X Programmer's Manual'. Set prevailing indent and tabs to .5i.
.TP *i*	yes	*i*=p.i.	Set prevailing indent to *i*. Begin indented paragraph with hanging tag given by next text line. If tag doesn't fit, place it on separate line.

* n.t.l. = next text line; p.i. = prevailing indent

NAME

me – macros for formatting papers

SYNOPSIS

nroff –me [options] file ...
troff –me [options] file ...

DESCRIPTION

This package of *nroff* and *troff* macro definitions provides a canned formatting facility for technical papers in various formats. When producing 2-column output on a terminal, filter the output through *col*(1).

The macro requests are defined below. Many *nroff* and *troff* requests are unsafe in conjunction with this package, however these requests may be used with impunity after the first **.pp**:

.bp	begin new page
.br	break output line here
.sp n	insert n spacing lines
.ls n	(line spacing) n=1 single, n=2 double space
.na	no alignment of right margin
.ce n	center next n lines
.ul n	underline next n lines
.sz +n	add n to point size

Output of the *eqn*, *neqn*, *refer*, and *tbl*(1) preprocessors for equations and tables is acceptable as input.

FILES

/usr/lib/tmac/tmac.e
/usr/lib/me/*

SEE ALSO

neqn(1), troff(1)(optional product), refer(1), tbl(1)
"–me Reference Manual" and "Writing Papers with Nroff Using –me" in the *ConvexOS Tutorial Papers*

REQUESTS

In the following list, "initialization" refers to the first .pp, .lp, .ip, .np, .sh, or .uh macro. This list is incomplete; see the "The –me Reference Manual" for interesting details.

Request	Initial Value	Cause Break	Explanation
.(c	-	yes	Begin centered block
.(d	-	no	Begin delayed text
.(f	-	no	Begin footnote
.(l	-	yes	Begin list
.(q	-	yes	Begin major quote
.(x *x*	-	no	Begin indexed item in index *x*
.(z	-	no	Begin floating keep
.)c	-	yes	End centered block
.)d	-	yes	End delayed text
.)f	-	yes	End footnote
.)l	-	yes	End list
.)q	-	yes	End major quote
.)x	-	yes	End index item
.)z	-	yes	End floating keep
.++ *m H*	-	no	Define paper section. *m* defines the part of the paper, and can be **C** (chapter), **A** (appendix), **P** (preliminary, e.g., abstract, table of contents, etc.), **B** (bibliography), **RC** (chapters renumbered from page one each chapter), or **RA** (appendix renumbered from page one).
.+c *T*	-	yes	Begin chapter (or appendix, etc., as set by .++). *T* is the chapter title.
.1c	1	yes	One column format on a new page.

.2c	1	yes	Two column format.
.EN	-	yes	Space after equation produced by *eqn* or *neqn*.
.EQ *x y*	-	yes	Precede equation; break out and add space. Equation number is *y*. The optional argument *x* may be *I* to indent equation (default), *L* to left-adjust the equation, or *C* to center the equation.
.TE	-	yes	End table.
.TH	-	yes	"" "" "ConvexOS V9.0 Programmer's Reference"
.TS *x*	-	yes	Begin table; if *x* is *H* table has repeated heading.
.ac *A N*	-	no	Set up for ACM style output. *A* is the Author's name(s), *N* is the total number of pages. Must be given before the first initialization.
.b *x*	no	no	Print *x* in boldface; if no argument switch to boldface.
.ba +*n*	0	yes	Augments the base indent by *n*. This indent is used to set the indent on regular text (like paragraphs).
.bc	no	yes	Begin new column
.bi *x*	no	no	Print *x* in bold italics (nofill only)
.bx *x*	no	no	Print *x* in a box (nofill only).
.ef *'x'y'z'*	''''	no	Set even footer to x y z
.eh *'x'y'z'*	''''	no	Set even header to x y z
.fo *'x'y'z'*	''''	no	Set footer to x y z
.hx	-	no	Suppress headers and footers on next page.
.he *'x'y'z'*	''''	no	Set header to x y z
.hl	-	yes	Draw a horizontal line
.i *x*	no	no	Italicize *x;* if *x* missing, italic text follows.
.ip *x y*	no	yes	Start indented paragraph, with hanging tag *x*. Indentation is *y* ens (default 5).
.lp	yes	yes	Start left-blocked paragraph.
.lo	-	no	Read in a file of local macros of the form .*x*. Must be given before initialization.
.np	1	yes	Start numbered paragraph.
.of *'x'y'z'*	''''	no	Set odd footer to x y z
.oh *'x'y'z'*	''''	no	Set odd header to x y z
.pd	-	yes	Print delayed text.
.pp	no	yes	Begin paragraph. First line indented.
.r	yes	no	Roman text follows.
.re	-	no	Reset tabs to default values.
.sc	no	no	Read in a file of special characters and diacritical marks. Must be given before initialization.
.sh *n x*	-	yes	Section head follows, font automatically bold. *n* is level of section, *x* is title of section.
.sk	no	no	Leave the next page blank. Only one page is remembered ahead.
.sz +*n*	10p	no	Augment the point size by *n* points.
.th	no	no	Produce the paper in thesis format. Must be given before initialization.
.tp	no	yes	Begin title page.
.u *x*	-	no	Underline argument (even in *troff*). (Nofill only).
.uh	-	yes	Like .sh but unnumbered.
.xp *x*	-	no	Print index *x*.

NAME

ms – text formatting macros

SYNOPSIS

nroff **–ms** [options] file ...
troff **–ms** [options] file ...

DESCRIPTION

This package of *nroff* and *troff* macro definitions provides a formatting facility for various styles of articles, theses, and books. When producing 2-column output on a terminal or lineprinter, or when reverse line motions are needed, filter the output through *col*(1). All external *-ms* macros are defined below. Many *nroff* and *troff* requests are unsafe in conjunction with this package. However, the first four requests below may be used with impunity after initialization, and the last two may be used even before initialization:

.bp	begin new page
.br	break output line
.sp n	insert n spacing lines
.ce n	center next n lines
.ls n	line spacing: n=1 single, n=2 double space
.na	no alignment of right margin

Font and point size changes with \f and \s are also allowed; for example, "\fIword\fR" will italicize *word*. Output of the *tbl*, *eqn*, and *refer*(1) preprocessors for equations, tables, and references is acceptable as input.

FILES

/usr/lib/tmac/tmac.x
/usr/lib/ms/x.???

SEE ALSO

neqn(1), refer(1), tbl(1), troff(1)(optional product)

REQUESTS

Macro Name	Initial Value	Break? Reset?	Explanation
.AB *x*	–	y	begin abstract; if *x*=no don't label abstract
.AE	–	y	end abstract
.AI	–	y	author's institution
.AM	–	n	better accent mark definitions
.AU	–	y	author's name
.B *x*	–	n	embolden *x*; if no *x*, switch to boldface
.B1	–	y	begin text to be enclosed in a box
.B2	–	y	end boxed text and print it
.BT	date	n	bottom title, printed at foot of page
.BX *x*	–	n	print word *x* in a box
.CM	if t	n	cut mark between pages
.CT	–	y,y	chapter title: page number moved to CF (TM only)
.DA *x*	if n	n	force date *x* at bottom of page; today if no *x*
.DE	–	y	end display (unfilled text) of any kind
.DS *x y*	I	y	begin display with keep; *x*=I,L,C,B; *y*=indent
.ID *y*	8n,.5i	y	indented display with no keep; *y*=indent
.LD	–	y	left display with no keep
.CD	–	y	centered display with no keep
.BD	–	y	block display; center entire block
.EF *x*	–	n	even page footer *x* (3 part as for .tl)
.EH *x*	–	n	even page header *x* (3 part as for .tl)
.EN	–	y	end displayed equation produced by *eqn*

.EQ *x y*	–	y	break out equation; *x*=L,I,C; *y*=equation number
.FE	–	n	end footnote to be placed at bottom of page
.FP	–	n	numbered footnote paragraph; may be redefined
.FS	–	n	start footnote to be placed at bottom of page
.HD	undef	n	optional page header below header margin
.I *x*	–	n	italicize *x*; if no *x*, switch to italics
.IP *x y*	–	y,y	indented paragraph, with hanging tag *x*; *y*=indent
.IX *x y*	–	y	index words *x y* and so on (up to 5 levels)
.KE	–	n	end keep of any kind
.KF	–	n	begin floating keep; text fills remainder of page
.KS	–	y	begin keep; unit kept together on a single page
.LG	–	n	larger; increase point size by 2
.LP	–	y,y	left (block) paragraph.
.MC *x*	–	y,y	multiple columns; *x*=column width
.ND *x*	if t	n	no date in page footer; *x* is date on cover
.NH *x y*	–	y,y	numbered header; *x*=level, *x*=0 resets, *x*=S sets to *y*
.NL	10p	n	set point size back to normal
.OF *x*	–	n	odd page footer *x* (3 part as for .tl)
.OH *x*	–	n	odd page header *x* (3 part as for .tl)
.P1	if TM	n	print header on 1st page
.PP	–	y,y	paragraph with first line indented
.PT	- -	n	page title, printed at head of page
.PX *x*	–	y	print index (table of contents); *x*=no suppresses title
.QP	–	y,y	quote paragraph (indented and shorter)
.R	on	n	return to Roman font
.RE	5n	y,y	retreat: end level of relative indentation
.RP *x*	–	n	released paper format; *x*=no stops title on 1st page
.RS	5n	y,y	right shift: start level of relative indentation
.SH	–	y,y	section header, in boldface
.SM	–	n	smaller; decrease point size by 2
.TA	8n,5n	n	set tabs to 8n 16n ... (nroff) 5n 10n ... (troff)
.TC *x*	–	y	print table of contents at end; *x*=no suppresses title
.TE	–	y	end of table processed by *tbl*
.TH	–	y	end multi-page header of table
.TL	–	y	title in boldface and two points larger
.TM	off	n	UC Berkeley thesis mode
.TS *x*	–	y,y	begin table; if *x*=H table has multi-page header
.UL *x*	–	n	underline *x*, even in *troff*
.UX *x*	–	n	UNIX; trademark message first time; *x* appended
.XA *x y*	–	y	another index entry; *x*=page or no for none; *y*=indent
.XE	–	y	end index entry (or series of .IX entries)
.XP	–	y,y	paragraph with first line exdented, others indented
.XS *x y*	–	y	begin index entry; *x*=page or no for none; *y*=indent
.1C	on	y,y	one column format, on a new page
.2C	–	y,y	begin two column format
.]-	–	n	beginning of *refer* reference
.[0	–	n	end of unclassifiable type of reference
.[N	–	n	N= 1:journal-article, 2:book, 3:book-article, 4:report

REGISTERS

Formatting distances can be controlled in *-ms* by means of built-in number registers. For example, this sets the line length to 6.5 inches:

```
.nr LL 6.5i
```

Here is a table of number registers and their default values:

Name	Register Controls	Takes Effect	Default
PS	point size	paragraph	10
VS	vertical spacing	paragraph	12
LL	line length	paragraph	6i
LT	title length	next page	same as LL
FL	footnote length	next .FS	5.5i
PD	paragraph distance	paragraph	1v (if n), .3v (if t)
DD	display distance	displays	1v (if n), .5v (if t)
PI	paragraph indent	paragraph	5n
QI	quote indent	next .QP	5n
FI	footnote indent	next .FS	2n
PO	page offset	next page	0 (if n), ~1i (if t)
HM	header margin	next page	1i
FM	footer margin	next page	1i
FF	footnote format	next .FS	0 (1, 2, 3 available)

When resetting these values, make sure to specify the appropriate units. Setting the line length to 7, for example, will result in output with one character per line. Setting FF to 1 suppresses footnote superscripting; setting it to 2 also suppresses indentation of the first line; and setting it to 3 produces an .IP-like footnote paragraph.

Here is a list of string registers available in *-ms*; they may be used anywhere in the text:

Name	String's Function
*Q	quote (" in *nroff*, `` in *troff*)
*U	unquote (" in *nroff*, '' in *troff*)
*–	dash (-- in *nroff*, — in *troff*)
*(MO	month (month of the year)
*(DY	day (current date)
*´	acute accent (before letter)
*`	grave accent (before letter)
*^	circumflex (before letter)
*,	cedilla (before letter)
*:	umlaut (before letter)
*~	tilde (before letter)

When using the extended accent mark definitions available with .AM, these strings should come after, rather than before, the letter to be accented.

BUGS

Floating keeps and regular keeps are diverted to the same space, so they cannot be mixed together with predictable results.

NAME

term – conventional names for terminals

DESCRIPTION

Certain commands use these terminal names. They are maintained as part of the shell environment (see *sh*(1),*environ*(7)).

adm3a	Lear Seigler Adm-3a
2621	Hewlett-Packard HP262? series terminals
hp	Hewlett-Packard HP264? series terminals
c100	Human Designed Systems Concept 100
h19	Heathkit H19
mime	Microterm mime in enhanced ACT IV mode
1620	DIABLO 1620 (and others using HyType II)
300	DASI/DTC/GSI 300 (and others using HyType I)
33	TELETYPE® Model 33
37	TELETYPE Model 37
43	TELETYPE Model 43
735	Texas Instruments TI735 (and TI725)
745	Texas Instruments TI745
dumb	terminals with no special features
dialup	a terminal on a phone line with no known characteristics
network	a terminal on a network connection with no known characteristics
4014	Tektronix 4014
vt52	Digital Equipment Corp. VT52

The list goes on and on. Consult /etc/termcap (see *termcap*(5)) for an up-to-date and locally correct list.

Commands whose behavior may depend on the terminal either consult TERM in the environment, or accept arguments of the form **–Tterm,** where *term* is one of the names given above.

SEE ALSO

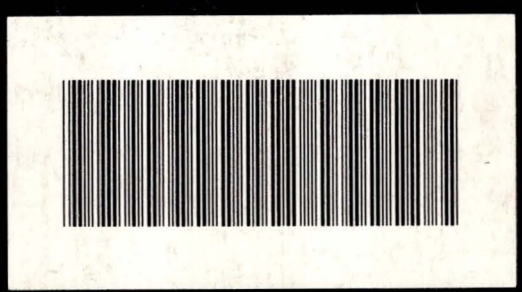
stty(1), tabs(1), sh(1), environ(7) ex(1), clear(1), more(1), ul(1), tset(1), termcap(5), termcap(3X), ttytype(5)
troff(1) for *nroff*

BUGS

The programs that ought to adhere to this nomenclature do so only fitfully.

Index

Description	Man Page
get processor time used	CLOCK(3)
get protocol entry	GETPROTOENT(3N)
get service entry	GETSERVENT(3N)
get socket name	GETSOCKNAME(2)
get system information	GETSYSINFO(2)
get system information	UNAME(2)
get system page size	GETPAGESIZE(2)
get system time	TIME(3C)
get terminal name	TTY(1)
get the current working directory for the process	GETCWD(3)
get ttys file entry	GETTTYENT(3)
get user identity	GETUID(2)
get user name	CUSERID(3)
getactaid — get activity file entry	GETACTENT(3)
getactent — get activity file entry	GETACTENT(3)
getactnam — get activity file entry	GETACTENT(3)
getacwent — get actwho file entry	GETACWENT(3)
getaid — get a process' activity ID	GETAID(2)
getc — get character or word from stream	GETC(3S)
getchar — get character or word from stream	GETC(3S)
getcwd — get the current working directory for the process	GETCWD(3)
getdirentries — gets directory entries in a filesystem independent format	GETDIRENTRIES(2)
getdiskbyname — get disk description by its name	GETDISKBYNAME(3X)
getdomainname — get/set name of current yellow pages domain	GETDOMAINNAME(2)
getdtablesize — get descriptor table size	GETDTABLESIZE(2)
getegid — get group identity	GETGID(2)
getenv — manipulate environmental variables	GETENV(3)
geteuid — get user identity	GETUID(2)
getfpmode — get current floating point mode	GETFPMODE(3)
getfsent — get file system descriptor file entry	GETFSENT(3X)
getfsfile — get file system descriptor file entry	GETFSENT(3X)
getfsspec — get file system descriptor file entry	GETFSENT(3X)
getfstype — get file system descriptor file entry	GETFSENT(3X)
getgid — get group identity	GETGID(2)
getgrent — get group file entry	GETGRENT(3)
getgrgid — group database access routines	GETGRGID(3)
getgrnam — group database access routines	GETGRGID(3)
getgroups — get group access list	GETGROUPS(2)
gethostbyaddr — get network host entry	GETHOSTBYNAME(3N)
gethostbyname — get network host entry	GETHOSTBYNAME(3N)
gethostent — get network host entry	GETHOSTBYNAME(3N)
gethostid — get/set unique identifier of current host	GETHOSTID(2)
gethostname — get/set name of current host	GETHOSTNAME(2)
getitimer — get/set value of interval timer	GETITIMER(2)
getlogin — get login name	GETLOGIN(3)
getlogin — get user name	CUSERID(3)
getmntent — get file system descriptor file entry	GETMNTENT(3)
getnetbyaddr — get network entry	GETNETENT(3N)



Order Number
DSW-331

Document Number
710-004030-003