NAME

cpio − copy file archives in and out

SYNOPSIS

cpio -o[Bv]
cpio -i[Bdmrtuv6] [pattern]
cpio -p[dlmruv] [pattern] directory

DESCRIPTION

Cpio −o (copy out) reads the standard input for a list of pathnames and copies those files onto the standard output together with pathname and status information.

Cpio −i (copy in) extracts from the standard input (which is assumed to be the product of a previous **cpio −o**) files whose names are selected by a *pattern* given in the name-generating syntax of *sh*(1). The *pattern* meta-characters ?, *, and [...] will match / characters. The *pattern* argument defaults to *.

Cpio −p (pass) copies out and in in a single operation. Destination pathnames are interpreted relative to the named *directory*.

The options are:

- **B** Input/output is blocked 5120 bytes to the record (does not apply to the *pass* option; meaningful only with data directed from/to **/dev/rmt?**).
- **d** *Directories* are to be created as needed.
- **r** Interactively *rename* files. If the user types a null line, the file is skipped.
- **t** Print a *table of contents* of the input. No files are created.
- **u** Copy *unconditionally* (normally, an older file will not replace a newer file with the same name).
- **v** *Verbose:* causes a list of file names to be printed. When used with the **t** option, the table of contents looks like an "ls −l" (see *ls*(1)).
- **l** Whenever possible, link files rather than copying them. Usable only with the **−p** option.
- **m** Retain previous file modification time. This option is ineffective on directories being copied.
- **6** Process an old (i.e., UNIX *Sixth* Edition file). Only useful with **−i** (copy in).

EXAMPLES

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

```
ls | cpio −o >/dev/mt0

cd olddir
find . −print | cpio −pdl newdir
```

SEE ALSO

ar(1), find(1)

BUGS

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

NAME

cpmv − copy move

SYNOPSIS

cpmv source mode uid gid [dest1 dest2 ...]

DESCRIPTION

The *cpmv* command is similar to the *move* command in that it copies the *source* file to the destination files and sets the mode, user id and group id (*mode*, *uid*, and *gid*, respectively) of the *destination* files. Unlike the *move* command, however, *cpmv* does not remove the *source* file.

FILES

/tmp/cpmv<*pid*>

SEE ALSO

mv(1), cp(1), move(1)

NAME

cref − make cross reference listing

SYNOPSIS

cref [**−acilnostux123**] name ...

DESCRIPTION

Cref makes a cross reference listing of program files in assembler or C format. The files named as arguments in the command line are searched for symbols in the appropriate syntax.

The output report is in four columns:

(1) (2) (3) (4)
symbol file see text as it appears in file
below

Cref uses either an *ignore* file or an *only* file. If the **−i** option is given, the next argument is taken to be an *ignore* file; if the **−o** option is given, the next argument is taken to be an *only* file. *Ignore* and *only* files are lists of symbols separated by new lines. All symbols in an *ignore* file are ignored in columns (1) and (3) of the output. If an *only* file is given, only symbols in that file appear in column (1). At most one of **−i** and **−o** may be used. The default setting is **−i**. Assembler predefined symbols or C keywords are ignored.

The **−s** option causes current symbols to be put in column 3. In the assembler, the current symbol is the most recent name symbol; in C, the current function name. The **−l** option causes the line number within the file to be put in column 3.

The **−t** option causes the next available argument to be used as the name of the intermediate temporary file (instead of /tmp/crt??). The file is created and is not removed at the end of the process.

Options:

a	assembler format (default)
c	C format input
i	use *ignore* file (see above)
l	put line number in col. 3 (instead of current symbol)
n	omit column 4 (no context)
o	use *only* file (see above)
s	current symbol in col. 3 (default)
t	user supplied temporary file
u	print only symbols that occur exactly once
x	print only C external symbols
1	sort output on column 1 (default)
2	sort output on column 2
3	sort output on column 3

FILES

/tmp/crt??	temporaries
/usr/lib/aign	default assembler *ignore* file
/usr/lib/atab	grammar table for assembler files
/usr/lib/cign	default C *ignore* file
/usr/lib/ctab	grammar table for C files
/usr/lib/crpost	post processor
/usr/lib/upost	post processor for **−u** option
/bin/sort	used to sort temporaries

SEE ALSO

as(1), cc(1), xref(1)

BUGS

Cref inserts "delete" characters into the intermediate temporary file after the eighth character of names which were eight or more characters in the source file.

NAME

cron − clock daemon

SYNOPSIS

/etc/cron

DESCRIPTION

Cron executes commands at specified dates and times according to the instructions in the file **/usr/lib/crontab** . *Cron* should be started from an entry in the lines file: see init(1M).

/usr/lib/crontab consists of lines of six fields separated by spaces or tabs. The first five are integer patterns to specify the minute (0-59), hour (0-23), day of the month (1-31), month of the year (1-12), and day of the week (0-6 with 0=Sunday). Each of these patterns may contain a number in the range above; two numbers separated by a minus (−) meaning a range inclusive; a list of numbers separated by commas (,) meaning any of the numbers; or an asterisk (*) meaning all legal values. The sixth field is a string that is executed by *sh* at the specified times. A percent (%) in this field is normally translated to a new-line character. Only the first line (up to a % or end of line) of the command field is executed by *sh* . The other lines are made available to the command as standard input. To escape this special meaning of %, immediately preceed the percent by a \. Similarily, to continue a line on subsequent lines the last character of the line should be a \. In this latter case, both the \ and the newline are discarded.

/usr/lib/crontab is examined by *cron* whenever *cron* is started, whenever *cron* is sent the interrupt signal and every hour on the hour. Thus it could take up to an hour for entries to become effective if *cron* is not signalled.

FILES

/usr/lib/crontab

SEE ALSO

init(1M), sh(1), kill(1)

DIAGNOSTICS

None − illegal lines in crontab are ignored.

NAME

crypt – encode/decode

SYNOPSIS

crypt key

DESCRIPTION

Crypt reads from the standard input and writes on the standard output. The argument is a key that selects a particular transformation. For any given key the transformation is idempotent; that is,

crypt key <clear >cypher
crypt key <cypher

will print the clear.

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

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

The transformation of a key into the internal settings of the machine is deliberately designed to be expensive, i.e. to take a substantial fraction of a second to compute. However, if keys are restricted to (say) three 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.

Crypt generates files which are compatible with the **-x** option in the editor.

SEE ALSO

ed(1)

NAME

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

SYNOPSIS

csh [-cefinstvVxX] [arg ...]

DESCRIPTION

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

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

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

Lexical structure

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

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

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

Commands

A simple command is a sequence of words, the first of which specifies the command to be executed. A simple command or a sequence of simple commands separated by `|' characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next. Sequences of pipelines may be

separated by `|', and are then executed sequentially. A sequence of pipelines may be executed without waiting for it to terminate by following it with an `&'. Such a sequence is automatically prevented from being terminated by a hangup signal; the nohup command need not be used.

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

Substitutions

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

History substitutions

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

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

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

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

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

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

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

string.

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

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

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

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

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

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

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

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

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

Quotations with ' and "

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

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

Alias substitution

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

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

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

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

Variable substitution

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

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

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

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

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

name, and the rest of which become arguments.

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

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

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

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

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

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

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

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

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

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

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

$?name
$<?name>
Substitutes the string `1' if name is set, `0' if it is not.

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

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

Command and filename substitution

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

Command substitution

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

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

Filename substitution

If a word contains any of the characters `*', `?', `[' or `{' or begins with the character `~', then that word is a candidate for

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

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

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

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

Input/output

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

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

<< word
: Read the shell input up to a line which is identical to word. Word is not subjected to variable, filename or command

substitution, and each input line is compared to word before any substitutions are done on this input line. Unless a quoting `\', `"', `'' or `\`' appears in word variable and command substitution is performed on the intervening lines, allowing `\' to quote `$', `\' and `\`'. Commands which are substituted have all blanks, tabs, and newlines preserved, except for the final newline which is dropped. The resultant text is placed in an anonymous temporary file which is given to the command as standard input.

> name
>! name
>& name
>&! name

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

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

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

>> name
>>& name
>>! name
>>&! name

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

If a command is run detached (followed by `&') then the default standard input for the command is the empty file `/dev/null'. Otherwise the command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Thus, unlike some previous shells, commands run from a file of shell commands have no access to the text of the commands by default; rather they receive the original standard input of the shell. The `<<' mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block read its input.

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

Expressions

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

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

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

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

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

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

Control flow

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

and, due to the implementation, restrict the placement of some of the commands.

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

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

Builtin commands

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

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

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

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

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

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

cd

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

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

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

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

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

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

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

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

foreach name (wordlist)
...
end
The variable name is successively set to each member of wordlist and the sequence of commands between this command

and the matching end are executed. (Both foreach and end must appear alone on separate lines.)

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

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

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

history
Displays the history event list.

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

```
if (expr) then
    ...
else if (expr2) then
    ...
else
    ...
endif
```

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

login
Terminate a login shell, replacing it with an instance of

/bin/login. This is one way to log off, included for compatibility with /bin/sh.

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

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

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

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

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

rehash
Causes the internal hash table of the contents of the directories in the path variable to be recomputed. This is needed if new commands are added to directories in the path while you are logged in. This should only be necessary if

you add commands to one of your own directories, or if a systems programmer changes the contents of one of the system directories.

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

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

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

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

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

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

switch (string)
case str1:
...
breaksw

...
default:
...
breaksw
endsw

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

time
time command

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

umask
umask value

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

unalias pattern

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

unhash

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

unset pattern

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

wait

All child processes are waited for. If the shell is

interactive, then an interrupt can disrupt the wait, at which time the shell prints names and process numbers of all children known to be outstanding.

while (expr)
...
end

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

@
@ name = expr
@ name[index] = expr

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

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

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

Pre-defined variables

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

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

argv Set to the arguments to the shell, it is from this variable that positional parameters are

substituted, i.e. `$1' is replaced by `$argv[1]', etc.

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

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

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

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

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

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

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

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

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

If multiple mail files are specified, then the

shell says 'New mail in name' when there is mail in the file name.

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

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

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

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

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

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

status — The status returned by the last command. If it

terminated abnormally, then 0200 is added to the status. Builtin commands which fail return exit status `1', all other builtin commands set status `0'.

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

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

Non-builtin command execution

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

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

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

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

Argument list processing

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

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

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

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

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

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

-s Command input is taken from the standard input.

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

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

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

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

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

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

Signal handling

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

AUTHOR

William Joy

FILES

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

LIMITATIONS

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

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

SEE ALSO

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

BUGS

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

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

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

Some commands should not touch status or it may be so transient

as to be almost useless. Oring in 0200 to status on abnormal termination is a kludge.

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

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

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

NAME

ct − call terminal

SYNOPSIS

ct [**−a**] [**−v**] [**−wn**] [**−s**speed] [telno]

DESCRIPTION

Ct dials the phone number of a modem that is attached to a terminal, and spawns a *getty* process for the specified terminal whose arg0 begins with a minus sign. *Telno* is the telephone number, with embedded **w**'s to wait for secondary dialtones.

Ct determines which dialers are associated with lines that are set to the appropriate speed by examining the file **/usr/lib/uucp/L-devices**. If all such available dialers are busy, *ct* will ask if it should wait for a line, and if so, for how many minutes it should wait before it gives up. *Ct* will continue to try to open the dialers at one-minute intervals until the specified limit is exceeded. The dialogue may be overridden by specifying the **−w***n* option, where *n* is the maximum number of minutes that *ct* is to wait to get a line.

Normally *ct* uses datasets with the class **ACU** as specified in the **L-devices** file. The **−a** flag causes *ct* to use datasets with the class **ACUA** instead. Such datasets are used to originate telephone calls using the answer tone and are therefore suitable for calling terminals such as silent 700s or couplers that cannot operate in the answer mode. *Ct* will also use the **ACUA** class if it finds that the environment variable **$L0** contains **−a**.

If the **−v** option is used, *ct* will send a running narrative to its error output.

The data rate may be set with the **−s** option, where *speed* is expressed in baud. The default rate is 300 baud.

FILES

/usr/lib/uucp/L-devices

SEE ALSO

cu(1C), getty(1M), login(1M), uucp(1C), dn(4)

NAME

cu − call another UNIX system

SYNOPSIS

cu telno [**−tneoi**] [**−s** speed] [**−anytilda**] [**−tandem**]

DESCRIPTION

Cu calls up another UNIX system, a terminal, or possibly a non-UNIX system. It manages an interactive conversation with possible transfers of text files. *Telno* is either the telephone number, with **w**'s at appropriate places to wait for secondary dialtone(s), a telephone number with the appropriate *uucp*(1C) dialcode prefixed, a system name listed in the *uucp* database (*L.sys*(5) file), or a hardwired line. *Cu* distinguishes all these various possibilities by looking at the telno character string. If no slashes appear and the first character is not a number, *cu* looks in the *uucp* database for information about the name. First *cu* accuses the telno string of being a system and looks in the *L.sys*(5) file of *uucp*. That failing, it tries to identify an initial string of alphabetic characters as being a location prefix as found in the *uucp* file *L-dialcodes*(5). That failing, the string is handed to the *conns*(3C) subroutine for an attempt to connect. If a system name was matched or a telephone prefix was found, *cu* generates a new teleno string (or if a hardwired line was found for the system in *L.sys*(5) the device name string is used). If the **−n** option is specified, *cu* prints the system name (if it exists), full telephone number and the line speed and exits. This useful when trying to determine whether you have guessed the correct system name or telephone location prefix.

The telno string is then handed to the *conns*(3C). subroutine which makes the actual connection to the remote system. If it fails, *cu* prints a message and exits.

The **−t** flag is used to dial out to a terminal. The **−anytilda** flag causes *cu* to accept the escape sequences listed below anywhere on a line, not just at the beginning (this is very useful when connecting to DEC ODT). Only the *send* process interprets the escape sequences anywhere on the line; the receive process is unaffected. The **−tandem** flag designates that the TANDEMI and TANDEMO flags are to be set meaning that XON/XOFF processing should take place. This allows *cu* to take and put files at 9600 baud. The **−e** (**−o**) flag designates that even (odd) parity is to be sent. If both **−e** and **−o** are on, marked parity is sent, ie. the high order bit is always set. This is useful when talking to the dataswitch. The **−i** switch puts *cu* into an interactive mode when selecting phone numbers from the *uucp* database. Since there may be more than one entry for a system, this is the only way to select some other entry than the first one encountered. When the **−i** switch is specified, *cu* will ask whether it should use each entry it finds in the *uucp* database. When you respond with **y**<*return*>, *cu* uses that number. Any other response and *cu* will continue looking in the database for another entry for the same system. *Speed* gives the transmission speed (110, 134, 150, 300, 1200, 4800, 9600); 300 is the default value.

After making the connection with the *conns*, *cu* runs as two processes: the *send* process reads the standard input and passes most of it to the remote system; the *receive* process reads from the remote system and passes most data to the standard output. Lines beginning with '~' have special meanings.

The *send* process interprets the following:

~.	terminate the conversation.
~EOT	terminate the conversation
~<*file*	send the contents of *file* to the remote system, as though typed at the terminal.
~!	invoke an interactive shell on the local system.
~!*cmd* ...	run *cmd* on the local system (via **sh −c**).

~$*cmd* ...	run *cmd* locally and send its output to the remote system.
~%take *from* [*to*]	copy file *from* (on the remote system) to file *to* on the local system. If *to* is omitted, the *from* name is used in both places.
~%put *from* [*to*]	copy file *from* (on local system) to file *to* on remote system. If *to* is omitted, the *from* name is used in both places.
~%cd *newdir*	change directory on local system.
~%speed *newspeed*	change the speed of the remote line.
~%anytilda	change the state of the *anytilda* flag to opposite. A "was [OFF | ON]" message is printed.
~%xclude	change the state of the XCLUDE bit for the remote line. A "was [OFF | ON]" message is printed.
~%tandem	change the state of XON/XOFF processing. A "was [OFF | ON]" message is printed.
~%break	send a break to the remote system.
~%?	print a list of wiggle (~) usages.
~~...	send the line ~... .

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

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

In any case, output is diverted (or appended, if >> is used) to the file. If : is used, the diversion is *silent*, i.e., it is written only to the file. If : is omitted, output is written both to the file and to the standard output. The trailing ~> terminates the diversion.

The use of **~%put** requires *stty*(1) and *cat*(1) on the remote side. It also requires that the current erase and kill characters on the remote system be identical to the current ones on the local system. Backslashes are inserted at appropriate places.

The use of **~%take** requires the existence of *echo* and *tee* on the remote system. Also, **stty tabs** mode is required on the remote system if tabs are to be copied without expansion.

FILES

/dev/null

SEE ALSO

cat(1), stty(1), uucp(1C), conns(3C), dh(4), dn(4), tty(4)

DIAGNOSTICS

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

BUGS

At speeds greater than 1200 baud, characters are likely to be lost unless the TANDEMI and TANDEMO bits are set by the option on the command line or through the wiggle escape sequence described above.
The algorithm used to send breaks is somewhat unreliable. The requirements for transfers (*stty*(1), *cat*(1), *echo*(1), and *tee*(1)) are not changeable.

NAME

cubic – three dimensional tic-tac-toe

SYNOPSIS

/usr/games/cubic

DESCRIPTION

Cubic plays the game of three dimensional 444 tic-tac-toe. Moves are given by the three digits (each 1-4) specifying the coordinate of the square to be played.

WARNING

Too much playing of the game will cause it to disappear.

NAME

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

SYNOPSIS

cut −clist [file1 file2 ...]
cut −flist [−dchar] [−s] [file1 file2 ...]

DESCRIPTION

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

The meanings of the options are:

list A comma-separated list of integer field numbers, with optional − to indicate ranges as in the **−o** option of *nroff/troff* for page ranges; e.g. "1,4,7" or "1−3,8" or "−5,10" (short for "1−5,10") or "3−" (short for third through last field).

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

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

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

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

Either **−c** or **−f** option must be specified.

EXAMPLES

ls −l | cut −c1,40−	lists current directory preceded by file type
cut −c−75 file ; cut −c76− file	to print a file with lines too long for terminal
cut −f1,7 table | tbl | nroff ...	prints columns 1 and 7 of *table*
cut −d: −f1,5 /etc/passwd	mapping of userids to names
name=`who am i | cut −f1 −d" "`	to set *name* to current userid

DIAGNOSTICS

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

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

no fields The *list* is empty.

SEE ALSO

grep(1) allows horizontal "cuts" (by context) through a file.
paste(1) allows to put files together columnwise, i. e. horizontally. To reorder columns in a table, use *cut* and *paste*.

NAME

date − print and set the date

SYNOPSIS

date [−s] [−v] [mmddhhmm[yy]] [+ format]

DESCRIPTION

If no argument is given, or if the argument begins with +, the current date and time are printed.

A numerical argument results in an attempt to set the system's idea of the current date. The argument is interpreted as follows:

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

date 10080045

sets the date to Oct 8, 12:45 AM. The current year is the default if no year is mentioned. The system operates in GMT. *Date* takes care of the conversion to and from local standard and daylight time.

If the -s option is used, *date* attempts to read a TCU100 battery powered clock and sets the system time to the clock time read.

The -v option makes date ask for verification before setting the time.

If the argument begins with +, the output format of *date* is under the control of the user. The format specification for the output is similar to that used in the first argument to *printf*(3S). All output fields are of fixed size (zero padded if necessary). Each field descriptor is preceded by % and will be replaced in the output by its corresponding value. A single % is encoded by %%. All other characters are copied to the output without change. The string is always terminated with a newline character.

Field Descriptors:

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

EXAMPLE

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

would generate as output:

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

DIAGNOSTICS

Most diagnostics are self-explanatory. Here are a few that aren't.

No permission if you aren't the super-user and you try to change the date;

bad conversion if the date set is syntactically incorrect;

invalid option if the field descriptor is not recognizable.

FILES

/dev/mem
/etc/wtmp

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 evaluated if they obey the stated relation.

v replaces the top element on the stack by its square root. Any existing fractional part of the argument is taken into account, but otherwise the scale factor is ignored.

! interprets the rest of the line as a UNIX command.

c All values on the stack are popped.

i The top value on the stack is popped and used as the number radix for further input. I pushes the input base on the top of the stack.

o The top value on the stack is popped and used as the number radix for further output.

O pushes the output base on the top of the stack.

k the top of the stack is popped, and that value is used as a non-negative scale factor: the appropriate number of places are printed on output, and maintained during multiplication, division, and exponentiation. The interaction of scale factor, input base, and output base will be reasonable if all are changed together.

z The stack level is pushed onto the stack.

Z replaces the number on the top of the stack with its length.

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

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

EXAMPLE

This example prints the first ten values of n!:

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

SEE ALSO

bc(1), which is a preprocessor for *dc* providing infix notation and a C-like syntax which implements functions and reasonable control structures for programs.

DIAGNOSTICS

x is unimplemented
where x is an octal number.

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

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

Out of headers
for too many numbers being kept around.

Out of pushdown
for too many items on the stack.

Nesting Depth
for too many levels of nested execution.

NAME

dcheck − file system directory consistency check

SYNOPSIS

dcheck [**−i** numbers] [filesystem]

DESCRIPTION

Dcheck reads the directories in a file system and compares the link-count in each i-node with the number of directory entries by which it is referenced. If the file system is not specified, a set of default file systems is checked.

The **−i** flag is followed by a list of i-numbers; when one of those i-numbers turns up in a directory, the number, the i-number of the directory, and the name of the entry are reported.

The program is fastest if the raw version of the special file is used, since the i-list is read in large chunks.

FILES

/dev/rrootdev Default file system.

DIAGNOSTICS

When a file turns up for which the link-count and the number of directory entries disagree, the relevant facts are reported. Allocated files which have 0 link-count and no entries are also listed. The only dangerous situation occurs when there are more entries than links; if entries are removed, so the link-count drops to 0, the remaining entries point to thin air. They should be removed. When there are more links than entries, or there is an allocated file with neither links nor entries, some disk space may be lost but the situation will not degenerate.

SEE ALSO

check(1M), clri(1M), ncheck(1M) fs(5),

BUGS

Since *dcheck* is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.

NAME

dd − convert and copy a file

SYNOPSIS

dd [option=value] ...

DESCRIPTION

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

option	*values*
if=*file*	input file name; standard input is default
of=*file*	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 in-core copy need be done
cbs=*n*	conversion buffer size
skip=*n*	skip *n* input records before starting copy
seek=*n*	seek *n* records from beginning of output file before copying
count=*n*	copy only *n* input records
conv=**ascii**	convert EBCDIC to ASCII
ebcdic	convert ASCII to EBCDIC
ibm	slightly different map of ASCII to EBCDIC
lcase	map alphabetics to lower case
ucase	map alphabetics to upper case
swab	swap every pair of bytes
noerror	do not stop processing on an error
sync	pad every input record to *ibs*
..., ...	several comma-separated conversions

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

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

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

EXAMPLE

This command will 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.

SEE ALSO

cp(1)

DIAGNOSTICS

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

BUGS

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

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

NAME

dead − crash analysis

SYNOPSIS

dead [**−smpcrbPMif**] crashfile [file]

DESCRIPTION

Dead produces formatted summaries of system tables from a crash dump. Rather than just print the contents of the system's tables, it attempts to put the information into a meaningful form while checking the consistency of pointers in the tables.

Dead can produce a memory map, a swap map, a list of the i-nodes that were open in the system (with fully qualified pathnames, including device name) and a list of the file descriptors that each process has open with the pathname of the corresponding file. In addition, it will analyze the block I/O buffering system to determine which buffers are on each device queue, which buffers are queued for I/O and which block devices are active. It can also retrieve the configuration table of the system.

All addresses in the system tables are printed in symbolic terms and the options can be used individually or in combination.

Dead assumes that the namelist for the crash dump is in "**/unix**" unless the s option is used, in which case a *file* name may be specified. There are several options:

c retrieves the configuration tables (conf.c) for character and block devices from the crash file. All entries in the tables are printed in symbolic form so that the printout closely resembles the contents of conf.c. The contents of the locations which specify the root and swap devices are also printed.

r prints the registers that were saved in low core by the crash dump routine when the dump was taken. The contents of the K register (KISA6 or KDSA6) contains the address (in memory blocks) of the last process that ran.

m prints a memory map and swap map with process names. Since reentrant text is managed separately from the nonreentrant part of a process, there can be two entries for a reentrant process. Reentrant text has a **T** appended to the address field. The following fields are printed,

Heading	*Description*
ADDR	The address of an area in memory blocks (64 bytes).
SIZE	The size of the area in memory blocks.
PID	The process id.
NAME	The name of the process occupying the area, or if the memory space is not allocated, "**Free**" is printed.

The swap map is also printed with the **m** option in a similar manner, however, the address and size fields are in disk blocks (512 bytes).

p prints a summary of all of the processes in the crash file, their names and attempts to construct a symbolic name for the event on which a process is roadblocked.

Heading	*Description*
NO	The number (index) of the process table entry.
S	A letter encoding the state of the process: S Sleeping W Waiting R Ready

I Idling

Z Zombie - process exited, parent not yet notified.

T Traced

F The location of the process. It may be any of the following:

D Swapped out

M In memory

L Locked in Memory

S Scheduler bit set

PID The unique process number.

PRI The priority of the process; high numbers mean low priority

UID Four characters of the user's id from the password file.

EVENT A symbolic representation of the address that the process is roadblocked on (if any).

NAME The file name of the process.

M prints the names of the mounted file systems and the pathnames of the i-nodes that they are mounted on.

i prints the contents of the *inode* table including an unambiguous pathname. The following fields are printed.

Heading *Description*

The number of the i-node table entry in decimal.

FLAGS Any special flags in the i-node are printed as follows,

D a directory

C a character device

B a block device

L a large file

ACCESS The access control permissions for the file are printed in a manner similar to the *ls* command. Permissions are printed in the same order as for *ls*, i.e., owner, group, others.

s set user-ID bit on

g set group-ID bit on

r read permission

w write permission

x execute permission

INO The number of the i-node.

UID The name of the owner of the file, taken from the password file.

DEVICE The pathname of the device on which the i-node resides.

PATHNAME

The pathname of the i-node if it can be found. Temporary files and pipes disappear when a system is rebooted, so some file names can not be found. Also, an i-node may be reallocated if a file is removed, so *dead* should be run relatively soon after a system is rebooted to insure that the pathnames are

correct.

f prints the *file* table.

Heading	*Description*
#	The number of the file table entry.
MD	The *mode* used to open the file.
C	The number of instances of the file being open with the given *mode*.
	R read permission
	W write permission
	P a pipe
	N a named pipe
DEVICE	The pathname of the device on which the file resides.
PATHNAME	The pathname of the i-node which the *file* table references.

P prints a list of all of the processes in memory and the files that they have open. The id of the process is printed with its name, followed by a list of file descriptors that the process has open and the pathnames of the files that they reference.

Heading	*Description*
#	The file descriptor number.
F	The *mode* with which the *file* was opened (same as MD field for the **f** option).
IND	The number of the *file* table entry which this descriptor references.
DEVICE	The pathname of the device on which the i-node resides.
PATHNAME	The pathname of the i-node.

b prints a summary of the buffers in the I/O subsystem and the queues on which they reside. There are five classes of queues. The *Buffer Free List* is a queue containing all of the buffers that are available for allocation. The *Null Device Queue* is a queue of buffers that are allocated by the system for some special purpose and are not associated with any device (e.g., holding superblocks, holding arguments for an *exec* system call, I/O for special devices, etc.). Each device analyzed by the **b** option has a number of queues but only those queues that are nonempty are printed. The *Device Queue* links together all of the buffers that contain data that have been retrieved or written onto the given device. The *Device Queue* may actually be a number of mashed subqueues. In this case, every subqueue will be shown. Buffers on this queue may also appear on the *Buffer Free List* . The *Device I/O Queue* contains all of the buffers that are actually waiting to be read or written from the given device. If a block device is active when a system crashes, the I/O queue is marked as *ACTIVE*. A symbolic representation of the pointers associated with each queue and each buffer on a queue is printed, however, familiarity with the I/O system is required to be able to check them and space does not permit such an explanation here. The values printed are,

Heading	*Description*
BUF	The buffer number.
FLAGS	Any of the following
	R Read
	W Write

D I/O complete (done)
B Buffer busy
E Error
P Physical I/O
U Unibus map allocated
N Wanted by other processes
A Asynchronous write
L Delayed write
G Age
H Physio Buffer Header
S State

[MAJ,MIN] Major and minor device numbers

BLOCK Logical block number

FILES

/unix	system namelist
unixcore	core image of unix crash
/etc/passwd	password file
/usr/bin/ncheck	

SEE ALSO

ncheck(1M), ps(1), sps(1)

BUGS

For the **P, M, i** and **f** options, *dead* runs *ncheck* to find the fully qualified pathname. This takes a bit of time.

NAME

dellog − print delta_log files

SYNOPSIS

dellog [opts] [directory ...]

DESCRIPTION

Dellog will print named delta_log files. The algorithm for finding the delta_log files is identical to that described in *file_log*(1S).

OPTIONS

all	print all delta_log files under $SCCSOURCE
list	print a list of all delta_log files under $SCCSOURCE

FILES

delta_log

SEE ALSO

file_log(1S), gadd(1S), gdelta(1S)

DIAGNOSTICS

All diagnostics are printed on file descriptor 2.

NAME

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

SYNOPSIS

delta [–rSID] [–s] [–n] [–glist] [–m[mrlist]] [–y[comment]] [–p] file ...

DESCRIPTION

Delta is used to permanently introduce into the named SCCS file changes that were made to the file retrieved by *get*(1S) (called the *g-file*, or generated file).

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

Delta may issue prompts on the standard output depending upon certain keyletters specified and flags (see *admin*(1S)) that may be present in the SCCS file (see **–m** and **–y** keyletters below).

Keyletter arguments apply independently to each named file.

–r*SID* Uniquely identifies which delta is to be made to the SCCS file. The use of this keyletter is necessary only if two or more outstanding *get*s for editing (*get* **–e**) on the same SCCS file were done by the same person (login name). The SID value specified with the **–r** keyletter can be either the SID specified on the *get* command line or the SID to be made as reported by the *get* command (see *get*). A diagnostic results if the specified SID is ambiguous, or, if necessary and omitted on the command line.

–s Suppresses the issue, on the standard output, of the created delta's SID, as well as the number of lines inserted, deleted and unchanged in the SCCS file.

–n Specifies retention of the edited *g-file* (normally removed at completion of delta processing).

–g*list* Specifies a *list* (see *get*(1S) for the definition of <*list*>) of deltas which are to be *ignored* when the file is accessed at the change level (SID) created by this delta.

–m[*mrlist*] If the SCCS file has the **v** flag set (see *admin*(1S)) then a Modification Request (MR) number *must* be supplied as the reason for creating the new delta.

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

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

Note that if the **v** flag has a value (see *admin*(1S)), it is taken to be the name of a program (or shell procedure) which will validate the correctness of the MR numbers. If a non-zero exit status is returned from MR number validation program, *delta* terminates (it is assumed that the MR numbers were not all valid).

—y[*comment*] Arbitrary text used to describe the reason for making the delta. A null string is considered a valid *comment.*

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

—p Causes *delta* to print (on the standard output) the SCCS file differences before and after the delta is applied in a *diff*(1S) format.

FILES

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

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

WARNINGS

No lines beginning with an SOH ASCII character (binary 001) can be placed in the SCCS file unless it is escaped. The character has special meaning to SCCS (see *sccsfile*(5)) and will cause an error.

A *get* of many SCCS files, followed by a *delta* of those files, should be avoided when the *get* generates a large amount of data. Instead, multiple *get/delta* sequences should be used.

If the standard input (—) is specified on the *delta* command line, the **—m** (if necessary) and **—y** keyletters *must* also be present. Omission of these keyletters causes an error to occur.

SEE ALSO

get(1S), admin(1S), prs(1S), help(1S), sccsfile(5), bdiff(1)
Source Code Control System User's Guide by L. E. Bonanni and C. A. Salemi.

DIAGNOSTICS

Use *help*(1S) for explanations.

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, (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), eqn(1), tbl(1)

NAME

df − disk free

SYNOPSIS

df [filesystem]

DESCRIPTION

Df prints out the number of free blocks available on the file system *filesystem.* If *filesystem* is unspecified, *df* uses a "built-in" list of file systems.

SEE ALSO

check(1), du(1)

BUGS

The "built-in" list of file systems is usually wrong so don't pay too much attention to what it tells you. *Df* should probably look at /etc/mtab to find out which file systems to examine.

NAME

diff − differential file comparator

SYNOPSIS

diff [**−efbh**] file1 file2

DESCRIPTION

Diff tells what lines must be changed in two files to bring them into agreement. If *file1* (*file2*) is −, the standard input is used. If *file1* (*file2*) is a directory, then a file in that directory whose file-name is the same as the file-name of *file2* (*file1*) is used. The normal output contains lines of these forms:

n1 a *n3,n4*
n1,n2 d *n3*
n1,n2 c *n3,n4*

These lines resemble *ed* commands to convert *file1* into *file2*. The numbers after the letters pertain to *file2*. In fact, by exchanging 'a' for 'd' and reading backward one may ascertain equally how to convert *file2* into *file1*. As in *ed*, identical pairs where *n1* = *n2* or *n3* = *n4* are abbreviated as a single number.

Following each of these lines come all the lines that are affected in the first file flagged by <, then all the lines that are affected in the second file flagged by >.

The **−b** option causes trailing blanks (spaces and tabs) to be ignored and other strings of blanks to compare equal.

The **−e** option produces a script of *a*, *c* and *d* commands for the editor *ed*, which will recreate *file2* from *file1*. The **−f** option produces a similar script, not useful with *ed*, in the opposite order. In connection with **−e**, the following shell program may help maintain multiple versions of a file. Only an ancestral file ($1) and a chain of version-to-version *ed* scripts ($2,$3,...) made by *diff* need be on hand. A 'latest version' appears on the standard output.

```
(shift; cat $*; echo '1,$p') | ed − $1
```

Except in rare circumstances, *diff* finds a smallest sufficient set of file differences.

Option **−h** does a fast, half-hearted job. It works only when changed stretches are short and well separated, but does work on files of unlimited length. Options **−e** and **−f** are unavailable with **−h**.

FILES

/tmp/d?????
/usr/lib/diffh for **−h**

SEE ALSO

cmp(1), comm(1), ed(1), bdiff(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 . .

NAME

diff3 − 3-way differential file comparison

SYNOPSIS

diff3 [−ex3] file1 file2 file3

DESCRIPTION

Diff3 compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

====	all three files differ
====1	*file1* is different
====2	*file2* is different
====3	*file3* is different

The type of change suffered in converting a given range of a given file to some other is indicated in one of these ways:

f : *n1* **a**	Text is to be appended after line number *n1* in file *f*, where *f* = 1, 2, or 3.
f : *n1* , *n2* **c**	Text is to be changed in the range line *n1* to line *n2*. If *n1* = *n2*, the range may be abbreviated to *n1*.

The original contents of the range follows immediately after a c indication. When the contents of two files are identical, the contents of the lower-numbered file is suppressed.

Under the −e option, *diff3* publishes a script for the editor *ed* that will incorporate into *file1* all changes between *file2* and *file3, i.e.* the changes that normally would be flagged ==== and ====3. Option −x (−3) produces a script to incorporate only changes flagged ==== (====3). The following command will apply the resulting script to *file1*.

```
(cat script; echo '1,$p') | ed − file1
```

FILES

/tmp/d3*
/usr/lib/diff3prog

SEE ALSO

diff(1)

BUGS

Text lines that consist of a single . will defeat −e.
Files longer than 64K bytes won't work.

NAME

diffmk — mark differences between files

SYNOPSIS

diffmk name1 name2 name3

DESCRIPTION

Diffmk compares two versions of a file and creates a third file that includes "change mark" commands for *nroff*(1) or *troff*(1). *Name1* and *name2* are the old and new versions of the file. *Diffmk* generates *name3*, which contains the lines of *name2* plus inserted formatter "change mark" (**.mc**) requests. When *name3* is formatted, changed or inserted text is shown by | at the right margin of each line. The position of deleted text is shown by a single *.

If anyone is so inclined, they can use *diffmk* to produce listings of C (or other) programs with changes marked. A typical command line is:

```
diffmk old.c new.c tmp; nroff macs tmp | pr
```

where the file **macs** contains:

```
.pl 1
.ll 77
.nf
.eo
.nc `
```

The **.ll** request might specify a different line length, depending on the nature of the program being printed. The **.eo** and **.nc** requests are probably needed only for C programs.

If the characters | and * are inappropriate, a copy of *diffmk* can be edited to change them (*diffmk* is a shell procedure).

SEE ALSO

diff(1), nroff(1).

BUGS

Aesthetic considerations may dictate manual adjustment of some output. File differences involving only formatting requests may produce undesirable output, i.e., replacing **.sp** by **.sp 2** will produce a "change mark" on the preceding or following line of output.

NAME

dircmp – directory comparison

SYNOPSIS

dircmp dir1 dir2

DESCRIPTION

Dircmp examines *dir1* and *dir2* and generates various tabulated information about the contents of the directories. Listings of files that are unique to each directory are generated in addition to a list that indicates whether the files common to both directories have the same contents.

SEE ALSO

cmp(1), diff(1)

NAME

dmplfs − dump logical file system to tape

SYNOPSIS

dmplfs lfs_name tape_unit#

DESCRIPTION

Dmplfs copies a logical file system (LFS) to tape. Unlike *dd*, which does a device-to-device copy, *dmplfs* writes two tape files, the first containing the LFS overhead area (header, file definition entries, freelist and bitmap), and the second the contents of all allocated logical files in ascending order. *Dmplfs* can be used to save the contents of the LFS for later restoral by *rstlfs*(1), or, because of the manner in which the files are dumped and restored, to compress an LFS whose free space has become highly fragmented (freelist full of areas too small to be used).

Lfs_name is the filename of the LFS in /dev and *tape_unit#* is the number of the tape drive on which the dump tape is mounted. Both parameters are required, and the program assumes that the tape is 2400 ft. long and will be written at 1600 bpi. For convenience, the user may specify the tape unit as 0-3; the program will modify the unit number as necessary to get the correct density. If the command is entered with no parameters, the program will print the expected syntax.

Dmplfs assumes that the overhead file will fit on one tape reel and that the data file may require more than one reel; the program will prompt the user when a new reel is to be mounted. An 80-character label file is written at the beginning of each reel (including the first) which contains the *lfs_name*, reel number, date and time. The blocking factor for both the overhead and data files is 5120 bytes (10 sectors) per tape block. In the overhead file, all tape blocks are full size (5120) except possibly the last block in the file, which may be shorter. In the data file, every logical file is written beginning on a tape block boundary, and if the file is less than 10 sectors long the tape block contains only the allocated file size in units of LF blocks. Similarly, the last tape block of a logical file contains only the remainder of the file in units of LF blocks.

FILES

/dev/*lfs_name*	LFS to be written to tape
/dev/mt*tape_unit#*	tape unit to be used
/etc/lmtab	list of mounted logical file systems

SEE ALSO

lfcheck(1), mklfs(1), rstlfs(1)

DIAGNOSTICS

Dmplfs prints self-explanatory error messages on exit whenever a problem is detected.

WARNINGS

Dmplfs uses the start and size information in the file definition entries to read the logical files from disk which can result in the "unfolding" of overlapped files (files containing duplicated blocks) as well as attempts to read overhead or bad blocks which have been erroneously allocated to files. These side effects can be prevented by making sure that the LFS checks (using *lfcheck*(1)) before dumping to tape.

Do not attempt to dump a mounted logical filesystem; the LFS should be unmounted and flushed to disk before *dmplfs*
is invoked.

The LFS should be re-made using *mklfs*(1) before restoring with *rstlfs*(1). As additional insurance, it is wise to make a dd tape of the LFS block device before doing the *mklfs* so the LFS can be restored to its prior state if necessary (i.e., if *rstlfs* has trouble reading the *dmplfs* tape).

Dmplfs assumes that the 1600 bpi tape units have file names **/dev/mt8** − **/dev/mt11** (rewind) and **/dev/mt12** − **/dev/mt15** (no rewind).

BUGS

In order to prevent the tape running off the end of the reel, there is an artificial limit of 6400 tape blocks per reel for the data file. This number was chosen to allow room for the drive to skip over bad spots on the tape when writing.

NAME

dsw - delete from switches

SYNOPSIS

(put number in console switches)
dsw
core

DESCRIPTION

dsw reads the console switches to obtain a number n, prints the name of the n-th file in the current directory, and exits, leaving a core image file named core. If this core file is executed, the file whose name was last printed is unlinked (see unlink(2)).

The command is useful for deleting files whose names are difficult to type.

SEE ALSO

rm(1), unlink(2)

BUGS

This command was written in 2 minutes to delete a particular file that managed to get an 0200 bit in its name. It should work by printing the name of each file in a specified directory and requesting a `y' or `n' answer. Better, it should be an option of rm(1).

The name is mnemonic, but likely to cause trouble in the future.

NAME

dsw – delete interactively

SYNOPSIS

dsw [directory]

DESCRIPTION

For each file in *directory* ('.' if not specified) *dsw* types the file name. If **y** is typed, the file is deleted; if **x**, *dsw* exits; if new-line, the file is not deleted; if anything else, *dsw* asks again.

SEE ALSO

rm(1)

BUGS

The name *dsw* is a carryover from the ancient past. Its etymology is amusing.

NAME

du – summarize disk usage

SYNOPSIS

du [–**ars**] [name ...]

DESCRIPTION

Du gives the number of blocks contained in all files and (recursively) directories within each specified directory or file *name*. If *name* is missing, . is used.

The optional argument –s causes only the grand total (for each *name*) to be given. The optional argument –**a** causes an entry to be generated for each file. Absence of either causes an entry to be generated for each directory only.

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

A file which has two links to it is only counted once.

BUGS

Non-directories given as arguments (not under –**a** option) are not listed.
If there are too many distinct linked files, *du* will count the excess files more than once. If directories are linked back on self, du will get into a loop and multiply report usages from the directories in the loop.

NAME

echo, fecho − echo arguments

SYNOPSIS

echo [arg] ...

fecho [arg] ...

DESCRIPTION

Echo and *fecho* write their arguments separated by blanks and terminated by a new-line on the standard output. *Fecho* is built into the shell, and is therefore much faster to execute than is the separate module *echo*. However, its output cannot be redirected as easily as that of *echo*.

Both commands understand C-like escape conventions; beware of conflicts with the shell's use of \:

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

In addition, a final argument which terminates in a blank will result in printing neither the blank nor the terminating newline. This is equivalent to the \c option, and is provided for compatibility with previous versions.

These commands are useful for producing diagnostics in command files and for sending known data into a pipe.

SEE ALSO

sh(1)

NAME

ed − text editor

SYNOPSIS

ed [−] [−x] [name]

DESCRIPTION

Ed is the standard text editor. If the *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. The optional − suppresses the printing of character counts by *e*, *r*, and *w* commands. If **−x** is present, an *x* command is simulated first to handle an encrypted file.

Ed operates on a copy of the file it is editing; changes made 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*. There is only one buffer for each invocation of*ed*.

Commands to *ed* have a simple and regular structure: zero, one, or two *addresses* followed by a single-character *command*, possibly followed by parameters to that command. These addresses specify one or more lines in the buffer. Every command that requires addresses has default addresses, so that the addresses can very often be omitted.

In general, only one command may appear on a line. Certain commands allow the input of text. This text is placed in the appropriate place in the buffer. While *ed* is accepting text, it is said to be in *input mode*. In this mode, *no* commands are recognized; all input is merely collected. Input mode is left by typing a period (.) alone at the beginning of a line.

Ed supports a limited form of *regular expression* notation; regular expressions are used in addresses to specify lines and in some commands (e.g., *s*) to specify portions of a line that are to be substituted. A regular expression (RE) specifies a set of character strings. A member of this set of strings is said to be *matched* by the RE. The REs allowed by *ed* are constructed as follows:

The following *one-character* REs match a *single* character:

1.1 An ordinary character (*not* one of those discussed in 1.2 below) is a one-character RE that matches itself.

1.2 A backslash (\) followed by any special character is a one-character RE that matches the special character itself. The special characters are:

a. ., *, [, and \ (period, asterisk, left square bracket, and backslash, respectively), which are always special, *except* when they appear within square brackets [ ] (see 1.4 below).

b. ^ (caret or circumflex), which is special at the *beginning* of an *entire* RE (see 3.1 and 3.2 below), or when it immediately follows the left of a pair of square brackets ([ ]) (see 1.4 below).

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

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

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

1.4 A non-empty string of characters enclosed in square brackets ([ ]) is a one-character RE that matches *any one* character in that string. If, however, the first character of the string is a circumflex (^), the one-character RE matches any character *except* new-line and the remaining characters in the string. The ^ has this special meaning *only* if it occurs first in the string. The minus (−) may be used to indicate a range of consecutive ASCII characters; for example, **[0−9]** is equivalent to **[0123456789]**. The − loses this special meaning

if it occurs first (after an initial ^, if any) or last in the string. The right square bracket (]) does not terminate such a string when it is the first character within it (after an initial ^, if any); e.g., []a−f] matches either a right square bracket (]) or one of the letters **a** through **f** inclusive. The four characters listed in 1.2.a above stand for themselves within such a string of characters.

The following rules may be used to construct REs from *one-character* REs:

2.1 A one-character RE is a RE that matches whatever the one-character RE matches.

2.2 A one-character RE followed by an asterisk (*) is a RE that matches *zero* or more occurrences of the one-character RE. If there is any choice, the longest leftmost string that permits a match is chosen.

2.3 A one-character RE followed by \{*m*\}, \{*m*,\}, or \{*m*,*n*\} is a RE that matches a *range* of occurrences of the one-character RE. The values of *m* and *n* must be non-negative integers less than 256; \{*m*\} matches *exactly m* occurrences; \{*m*,\} matches *at least m* occurrences; \{*m*,*n*\} matches *any number* of occurrences *between m* and *n* inclusive. Whenever a choice exists, the RE matches as many occurrences as possible.

2.4 The concatenation of REs is a RE that matches the concatenation of the strings matched by each component of the RE.

2.5 A RE enclosed between the character sequences \(and \) is a RE that matches whatever the unadorned RE matches.

2.6 The expression *n* matches the same string of characters as was matched by an expression enclosed between \(and \) *earlier* in the same RE. Here *n* is a digit; the sub-expression specified is that beginning with the *n*-th occurrence of \(counting from the left. For example, the expression ^\(.*\)\1$ matches a line consisting of two repeated appearances of the same string.

Finally, an *entire* RE may be constrained to match only an initial segment or final segment of a line (or both):

3.1 A circumflex (^) at the beginning of an entire RE constrains that RE to match an *initial* segment of a line.

3.2 A currency symbol ($) at the end of an entire RE constrains that RE to match a *final* segment of a line. The construction ^*entire RE*$ constrains the entire RE to match the entire line.

The null RE standing alone (e.g., //) is equivalent to the last RE encountered.

To understand addressing in *ed* it is necessary to know that at any time there is a *current line*. Generally speaking, the current line is the last line affected by a command; the exact effect on the current line is discussed under the description of each command. Addresses are constructed as follows:

1. The character . addresses the current line.
2. The character $ addresses the last line of the buffer.
3. A decimal number *n* addresses the *n*-th line of the buffer.
4. 'x addresses the line marked with the mark name character *x*, which must be a lower-case letter. Lines are marked with the *k* command described below.
5. A RE enclosed by slashes (/) addresses the first line found by searching *forward* from the line *following* the current line toward the end of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the beginning of the buffer and continues up to and including the current line, so that the entire buffer is searched.

6. A RE enclosed in question marks (?) addresses the first line found by searching *backward* from the line *preceding* the current line toward the beginning of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the end of the buffer and continues up to and including the current line.
7. An address followed by a plus sign (+) or a minus sign (−) followed by a decimal number specifies that address plus (respectively minus) the indicated number of lines. The plus sign may be omitted.
8. If an address begins with + or −, the addition or subtraction is taken with respect to the current line; e.g, **−5** is understood to mean **.−5**.
9. If an address ends with + or −, then 1 is added to or subtracted from the address, respectively. As a consequence of this rule and of rule 8 immediately above, the address − refers to the line preceding the current line. (To maintain compatibility with earlier versions of the editor, the character ^ in addresses is entirely equivalent to −.) Moreover, trailing + and − characters have a cumulative effect, so −− refers to the current line less 2.
10. For convenience, a comma (,) stands for the address pair **1,$**, while a semicolon (;) stands for the pair **.,$**.

Commands may require zero, one, or two addresses. Commands that require no addresses regard the presence of an address as an error. Commands that accept one or two addresses assume default addresses when an insufficient number of addresses is given; if more addresses are given than such a command requires, the last one(s) are used.

Typically, addresses are separated from each other by a comma (,). They may also be separated by a semicolon (;). In the latter case, the current line (.) is set to the first address, and only then is the second address calculated. This feature can be used to determine the starting line for forward and backward searches (see rules 5. and 6. above). The second address of any two-address sequence must correspond to a line that follows, in the buffer, the line corresponding to the first address.

In the following list of *ed* commands, the default addresses are shown in parentheses. The parentheses are *not* part of the address; they show that the given addresses are the default.

It is generally illegal for more than one command to appear on a line. However, any command (except *e*, *f*, *r*, or *w*) may be suffixed by **p** or by **l**, in which case the current line is either printed or listed, respectively, as discussed below under the *p* and *l* commands.

(.)a
<text>
.

The *a*ppend command reads the given text and appends it after the addressed line; . is left at the last inserted line, or, if there were none, at the addressed line. Address 0 is legal for this command: it causes the "appended" text to be placed at the beginning of the buffer.

(.)c
<text>
.

The *c*hange command deletes the addressed lines, then accepts input text that replaces these lines; . is left at the last line input, or, if there were none, at the first line that was not deleted.

(.,.)d

The *d*elete command deletes the addressed lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end of the buffer, the new last line becomes the current line.

e *name*

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

E *name*

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

f *name*

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

(1,$)g/*RE***/***command list*

In the *g*lobal command, the first step is to mark every line that matches the given RE. Then, for every such line, the given *command list* is executed with . initially set to that line. A single command or the first of a list of commands appears on the same line as the global command. All lines of a multi-line list except the last line must be ended with a \; *a*, *i*, and *c* commands and associated input are permitted; the . terminating input mode may be omitted if it would be the last line of the *command list*. The (global) commands (*g*, *G*, *v*, and *V*) are *not* permitted in the *command list*.

(1,$)G/*RE***/**

In the interactive *G*lobal command, the first step is to mark every line that matches the given RE. Then, for every such line, that line is printed, . is changed to that line, and any *one* command (other than one of the global commands *q*, *G*, *v*, and *V*) may be input and is executed. After the execution of that command, the next marked line is printed, and so on; a new-line acts as a null command; an & causes the re-execution of the most recent command executed within the current invocation of *G*. Note that the commands input as part of the execution of the *G* command may address and affect *any* lines in the buffer. The *G* command can be terminated by an interrupt signal (ASCII DEL or BREAK).

h

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

H

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

(.)i
<text>
.

The *i*nsert command inserts the given text before the addressed line; . is left at the last inserted line, or, if there were none, at the addressed line. This command differs from the *a* command only in the placement of the input text. Address 0 is not legal for this command.

(.,.+1)j

The *j*oin command joins contiguous lines by removing the appropriate new-line characters. If only one address is given, this command does nothing.

(.)k*x*

The mar*k* command marks the addressed line with name *x*, which must be a lower-case letter. The address ´*x* then addresses this line; . is unchanged.

(.,.)l

The *l*ist command prints the addressed lines in an unambiguous way: a few non-printing characters (e.g., *tab, backspace*) are represented by (hopefully) mnemonic overstrikes, all other non-printing characters are printed in octal, and long lines are folded. An *l* command may be appended to any other command other than *e*, *f*, *r*, or *w*.

(.,.)m*a*

The *m*ove command repositions the addressed line(s) after the line addressed by *a*. Address 0 is legal for *a* and causes the addressed line(s) to be moved to the beginning of the file; it is an error if address *a* falls within the range of moved lines; . is left at the last line moved.

(.,.)p

The *p*rint command prints the addressed lines; . is left at the last line printed. The *p* command may be appended to any other command *e*, *f*, *r*, or *w*; for example, *dp* deletes the current line and prints the new current line.

P

The editor will prompt with a * for all subsequent commands. The *P* command alternately turns this mode on and off; it is initially off.

q

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

Q

The editor exits without checking if changes have been made in the buffer since the last *w* command.

($)r *name*

The *r*ead command reads in the given file after the addressed line. If no file name is given, the currently-remembered file name, if any, is used (see *e* and *f* commands). The currently-remembered file name is not changed unless *name* is the very first file name mentioned since *ed* was invoked. Address 0 is legal for *r* and causes the file to be read at the beginning of the buffer. If the read is successful, the number of characters read is typed; . is set to the last line read in. If the name used in an *r* command begins with !, the rest of the line is taken to be a shell (*sh*(1)) command to be read from. Such a command is *not* remembered as the current file name.

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

The *s*ubstitute command searches each addressed line for an occurrence of the specified regular expression. On each line in which a match is found, matched strings are replaced under control of *range*. *Range* can appear in one of four ways. If it is empty only the first occurence of the matched string is replaced. If *range* is "g", all matches are replaced. If a single number appears only the match that number from the left is replaced. If a pair of numbers separated by a "," appears, the first is a starting point and the second is a count. No error occurs if the number of matches is less than the second number. Instead of a number, a "$" may be used to refer to the last possible match on a line. It is an error for the substitution to fail on all addressed lines. Any character other than space or new-line may be used instead of "/" to delimit the regular expression and the replacement. "." is left at the last line substituted.