Example

echo — Command

Repeat/expand an argument
echo [-n] [*argument ...*]

echo prints each *argument* on the standard output, placing a space between each *argument*. It appends a newline to the end of the output unless the **-n** flag is present.

echo recognizes the following special character sequences. For each occurrence of the sequence, it substitutes the corresponding ASCII character.

\b	Backspace
\c	Print line without a newline (like **-n** option)
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Tab
\v	Vertical tab
\\	Backslash
\0*nnn*	*nnn* is octal value of character (**sh** only)
nnn	*nnn* is the octal value of character (**ksh** only)

For example, when you enter the command:

```
echo 'Please enter your name: \007\c'
```

The shell rings the bell and prints

```
Please enter your name:
```

on your screen. Note that the **\007** sequence causes the terminal bell to sound, and that since the **\c** sequence was specified, the cursor will be left positioned after the colon.

See Also

commands, ksh, sh

Notes

Under the Korn shell, **echo** is an alias for its built-in command **print**.

Please note that **echo** converts characters to spaces. If you wish to preseve tab characters in an echoed string, you must enclose it within quotation marks. For example, the command

```
echo $RECORD
```

displays:

```
7 5 175 875
```

whereas the command

```
echo "$RECORD"
```

displays:

```
7     5     175   875
```

This is important when you use **echo** with programs for which the tab character is significant.

ecvt() — General Function (libc)

Convert floating-point numbers to strings
char *
ecvt(*d*, *prec*, *dp*, *signp*)
double *d*; int *prec*, **dp*, *signp;

ecvt() converts *d* into a NUL-terminated string of numerals with the precision of *prec*. Its operation resembles that of **printf()**'s operator **%e**.

ecvt() rounds the last digit and returns a pointer to the result. On return, **ecvt()** sets *dp* to point to an integer that indicates the location of the decimal point relative to the beginning of the string, to the right if positive, to the left if negative. It sets *signp* to point to an integer that indicates the sign of *d*, zero if positive and nonzero if negative.

Example

The following program demonstrates **ecvt()**, **fcvt()**, and **gcvt()**.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* prototypes for extended functions */
extern char *ecvt();
extern char *fcvt();
extern char *gcvt();

main(void)
{
      char buf[64];
      double d;
      int i, j;
      char *s;

      d = 1234.56789;
      s = ecvt(d, 5, &i, &j);
      /* prints ecvt="12346" i=4 j=0 */
      printf("ecvt=\"%s\" i=%d j=%d\n", s, i, j);

      strcpy(s, fcvt(d, 5, &i, &j));
      /* prints fcvt="123456789" i=4 j=0 */
      printf("fcvt=\"%s\" i=%d j=%d\n", s, i, j);

      s = gcvt(d, 5, buf);
      /* prints gcvt="1234.56789" */
      printf("gcvt=\"%s\"\n", s);
}
```

See Also

libc

Notes

ecvt() performs conversions within static string buffers that it overwrites with each execution.

ed — Command

Interactive line editor
ed [-] [+cmopsv] [*file*]

ed is the COHERENT system's interactive line editor.

ed is a line-oriented interactive text editor. With it, you can locate and replace text patterns, move or copy blocks of text, and print parts of the text. **ed** can read text from input files and can write all or part of the edited text to other files.

ed reads commands from the standard input, usually one command per line. Normally, **ed** does not prompt for commands. If the optional *file* argument is given, **ed** edits the given file, as if the *file* were read with the **e** command described below.

ed manipulates a copy of the text in memory rather than with the file itself. No changes to a file occur until the user writes edited text with the **w** command. Large files can be divided with **split** or edited with the stream editor **sed**.

ed remembers some information to simplify its commands. The *current line* is typically the line most recently edited or printed. When **ed** reads in a file, the last line read becomes the current line. The *current file name* is the last file name specified in an **e** or **f** command. The *current search pattern* is the last pattern specified in a search specification.

ed identifies text lines by integer line numbers, beginning with one for the first line. Several special forms identify a line or a range of lines, as follows:

n A decimal number *n* specifies the *n*th line of the text.

. A period '.' specifies the current line.

**$** A dollar sign '$' specifies the last line of the text.

+,- Simple arithmetic may be performed on line numbers.

/*pattern***/**
Search forward from the current line for the next occurrence of the *pattern*. If **ed** finds no occurrence before the end of the text, the search wraps to the beginning of the text. Patterns, also called *regular expressions*, are described in detail below.

?*pattern***?**
Search backwards from the current line to the previous occurrence of the *pattern*. If **ed** finds no occurrence before the beginning of the text, the search wraps to the end of the text.

'*x* Lines marked with the **k***x* command described below are identified by *'x*. The *x* may be any lower-case letter.

*n***,***m* Line specifiers separated by a comma ',' specify the range of lines between the two given lines, inclusive.

*n***;***m* Line specifiers separated by a semicolon ';' specify the range of lines between the two given lines, inclusive. Normally, **ed** updates the current line after it executes each command. If a semicolon ';' rather than a comma separates two line specifiers, **ed** updates the current line before reading the second.

***** An asterisk '*' specifies all lines; it is equivalent to **1,$**.

Commands

ed commands consist of a single letter, which may be preceded by one or two specifiers that give the line or lines to which the command is to be applied. The following command summary uses the notations [**n**] and [**n**[,**m**]] to refer to an optional line specifier and an optional range, respectively. These default to the current line when omitted, except where otherwise noted. A semicolon ';' may be used instead of a comma ',' to separate two line specifiers.

. Print the current line. Also, a line containing only a period '.' marks the end of **a**ppended, **c**hanged, or **i**nserted text.

[*n***]** Print given line. If no line number is given (i.e., the command line consists only of a newline character), print the line that follows the current line.

[*n***]=** Print the specified line number (default: last line number).

[*n***]&** Print a screen of 23 lines; equivalent to **n**,**n**+*22p*.

! *line* Pass the given *line* to the shell **sh** for execution. **ed** prompts with an exclamation point '!' when execution is completed.

? Print a brief description of the most recent error.

[*n***]a** Append new text after line *n*. Terminate new text with line that contains only a period '.'.

[*n***[,***m***]]c**
Change specified lines to new text. Terminate new text with a line that contains only a period '.'.

[*n***[,***m***]]d[p]**
Delete specified lines. If **p** follows, print new current line.

e [*file***]** Edit the specified *file* (default: current file name). An error occurs if there are unsaved changes. Reissuing the command after the error message forces **ed** to edit the *file*.

E [*file***]** Edit the specified *file* (default: current file name). No error occurs if there are unsaved changes.

f [*file***]** Change the current file name to *file* and print it. If *file* is omitted, print the current file name.

[*n***[,***m***]]g/[***pattern***]/***commands*
Globally execute *commands* for each line in the specified range (default: all lines) that contains the *pattern* (default: current search pattern). The *commands* may extend over several lines, with all but the last terminated by '\'.

[*n***]i** Insert text before line *n*. Terminate new text with a line that contains only a period '.'.

[*n***[,***m***]]j[p]**
Join specified lines into one line. If *m* is not specified, use range *n*,*n*+1. If no range is specified, join the current line with the next line. With optional **p**, print resulting line.

[*n***]k***x* Mark given line with lower-case letter *x*.

[*n***[,***m***]]l** List selected lines, interpreting non-graphic characters.

[*n***[,***m***]]m[***d***]**
Move selected lines to follow line *d* (default: current line).

o *options*
Change the given *options*. The *options* may consist of an optional sign '+' or '-', followed by one or more of the letters '*cmopsv*'. Options are explained below.

[*n***[,***m***]][p]**
Print selected lines. The **p** is optional.

q Quit editing and exit. An error occurs if there are unsaved changes. Reissuing the command after the error message forces **ed** to exit.

Q Quit editing and exit. Throw away all changes that you have not yet saved to disk.

[*n***]r [***file***]**
Read *file* into current text after given line (default: last line).

[n[,*m***]]s[***k***]/[***pattern1***]/***pattern2***/[g][p]**
Search for *pattern1* (default, remembered search pattern) and substitute *pattern2* for *k*th occurrence (default, first) on each line of the given range. If **g** follows, substitute every occurrence on each line. If **p** follows, print the resulting current line.

[*n***[,***m***]]t[***d***]**
Transfer (copy) selected lines to follow line *d* (default, current line).

[*n***]u[p]** Undo effect of last substitute command. If optional **p** specified, print undone line. The specified line must be the last substituted line.

[*n***[,***m***]]v/[***pattern***]/***commands*
Globally execute *commands* for each line in the specified range (default: all lines) *not* containing the *pattern* (default: current search pattern). The *commands* may extend over several lines, with all but the last terminated by '\'. The **v** command is like the **g** command, except the sense of the search is reversed.

[*n***[,***m***]]w [***file***]**
Write selected lines (default, all lines) to *file* (default, current file name). The previous contents of *file*, if any, are lost.

[*n***[,***m***]]W [***file***]**
Write specified lines (default, all lines) to the end of *file* (default, current file name). Like **w**, but appends to *file* instead of truncating it.

Patterns

Substitution commands and search specifications may include *patterns*, also called *regular expressions*. A non-special character in a pattern matches itself. Special characters include the following.

^ Match beginning of line, unless it appears immediately after '[' (see below).

$ Match end of line.

***** Matches zero or more repetitions of preceding character.

. Matches any character except newline.

[*chars***]** Matches any one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

[^*chars***]**
Matches any character *except* one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

****_c_ Disregard special meaning of character _c_.

\\(_pattern_**\\)**
Delimit substring _pattern_ for use with **\\d**, described below.

The replacement part _pattern2_ of the substitute command may also use the following:

& Insert characters matched by _pattern1_.

d Insert substring delimited by _d_th occurrence of delimiters '\\(' and '\\)', where _d_ is a digit.

Options

The user may specify **ed** options on the command line, in the environment, or with the **o** command. The available options are as follows:

c Print character counts on **e**, **r**, and **w** commands.

m Allow multiple commands per line.

o Print line counts instead of character counts on **e**, **r**, and **w** commands.

p Prompt with an '*' for each command.

s Match lower-case letters in a _pattern_ to both upper-case and lower-case text characters.

v Print verbose versions of error messages.

The **c** option is normally set, and all others are normally reset. Options may be set on the command line with a leading '+' sign. The '-' command line option resets the **c** option.

Options may be set in the environment with an assignment, such as

```
export ED=+cv
```

Options may be set with the '+' prefix or reset with the '-' prefix.

See Also

commands, elvis, ex, me, sed, vi
Introduction to the ed Line Editor

Diagnostics

ed usually prints only the diagnostic '?' on any error. When the verbose option **v** is specified, the '?' is followed by a brief description of the nature of the error.

EDITOR — Environmental Variable

Name editor to use by default
EDITOR=_editor_

The environmental variable **EDITOR** names the default editor that you wish to use. For example, **mail** invokes _editor_ when you conclude a mail message by typing a question mark '?' at the beginning of a line followed by **<return>**. The screen pager **more** invokes _editor_ when you enter the command **v** while displaying a file.

See Also

environmental variables, mail, more

egrep — Command

Extended pattern search
egrep [-Abcefhily] [_pattern_**] [**_file ..._**]**

egrep is an extended and faster version of **grep**. It searches each _file_ for occurrences of _pattern_ (also called a regular expression). If no _file_ is specified, it searches the standard input. Normally, it prints each line matching the _pattern_.

Wildcards

The simplest _patterns_ accepted by **egrep** are ordinary alphanumeric strings. Like **ed**, **egrep** can also process _patterns_ that include the following wildcard characters:

^ Match beginning of line, unless it appears immediately after '[' (see below).

$ Match end of line.

* Match zero or more repetitions of preceding character.

. Match any character except newline.

[*chars***]**
Match any one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

[^*chars***]**
Match any character *except* one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

*****c* Disregard special meaning of character *c*.

Metacharacters

In addition, **egrep** accepts the following additional metacharacters:

| Match the preceding pattern *or* the following pattern. For example, the pattern **cat|dog** matches either **cat** or **dog**. A newline within the *pattern* has the same meaning as '|'.

+ Match one or more occurrences of the immediately preceding pattern element; it works like '*', except it matches at least one occurrence instead of zero or more occurrences.

? Match zero or one occurrence of the preceding element of the pattern.

(...) Parentheses may be used to group patterns. For example, **(Ivan)+** matches a sequence of one or more occurrences of the four letters 'I' 'v' 'a' or 'n'.

Because the metacharacters '*', '?', '$', '(', ')', '[', ']', and '|' are also special to the shell, patterns that contain those literal characters must be quoted by enclosing *pattern* within apostrophes.

Options

The following lists the available options:

-A Write all lines in which *expression* is found into a temporary file. Then, call COHERENT with its error option to process the source file, with the contents of the temporary file serving as an "error" list. This option resembles the **-A** option to the **cc** command, and lets you build a COHERENT script to make systematic changes to the source file. To exit COHERENT and prevent **egrep** from searching further, **<ctrl-U> <ctrl-X> <ctrl-C>**.

Unlike **cgrep**, **egrep** only matches patterns that are on a single line. Some systems have a context **grep cgrep**) that works like **egrep** but displays lines found in context. The COHERENT **egrep -A** not only displays lines in context, via COHERENT, it lets you edit them.

-b With each output line, print the block number in which the line started (used to search file systems).

-c Print how many lines match, rather than the lines themselves.

-e The next argument is *pattern* (useful if the pattern starts with '-').

-f The next argument is a file that contains a list of patterns separated by newlines; there is no *pattern* argument.

-h When more than one *file* is specified, output lines are normally accompanied by the file name; **-h** suppresses this.

-i Ignore case when matches alphabetic letter in *pattern*. **egrep** takes case into account, even with this option, when you prefix a letter in *pattern* with '\\'.

-l Print the name of each file that contains the string, rather than the lines themselves. This is useful when you are constructing a batch file.

-n When a line is printed, also print its number within the file.

-s Suppress all output, just return exit status.

-v Print a line only if the pattern is *not* found in the line.

-y Lower-case letters in the pattern match only upper-case letters on the input lines. A letter escaped with '\' in the pattern must be matched in exactly that case.

Limits

The COHERENT implementation of **egrep** sets the following limits on input and output:

Characters per input record	512
Characters per output record	512
Characters per field	512

See Also

awk, cgrep, commands, ed, expr, grep, lex, sed

Diagnostics

egrep returns an exit status of zero for success, one for no matches, and two for error.

Notes

For matching patterns in C programs, the command **cgrep** is preferred, because it is optimized to recognize C-style expressions.

Besides the difference in the range of patterns allowed, **egrep** uses a deterministic finite automaton (DFA) for the search. It builds the DFA dynamically, so it begins doing useful work immediately. This means that **egrep** is is much faster than **grep**, often by more than an order of magnitude, and is considerably faster than earlier pattern-searching commands, on almost any length of file.

else — C Keyword

Introduce a conditional statement

else is the flip side of an **if** statement: if the condition described in the **if** statement fails, then the statements introduced by the **else** statement are executed. For example,

```
if (getchar() == EOF)
      exit(0);
else
      dosomething();
```

exits if the user types **EOF**, but does something if the user types anything else.

See Also

C keywords, if
ANSI Standard, §6.6.4.1

elvis — Command

Clone of Berkeley-standard screen editor
elvis [*options* **] [+***cmd* **] [** *file1 ... file27* **]**

elvis is a clone of **vi** and **ex**, the standard UNIX screen editors.

elvis is a modal editor whose command structure resembles the **ed** line editor. *Modal* means that a keystroke assumes a different meaning, depending upon the mode that the editor is in. **elvis** uses three modes: *visual-command mode, colon-command mode,* and *input mode.*

The following sections summarize the commands associated with each mode:

Visual-Command Mode

Visual-command mode closely resembles text-input mode. One quick way to tell the modes apart is to press the **<esc>** key. If **elvis** beeps, then you are in visual-command mode. If it does not beep, then you were in input mode, but pressing **<esc>** switched you to visual-command mode.

Most visual-mode commands are one keystroke long. The commands are in two groups: movement commands and edit commands. The former group moves the cursor through the file being edited, and the latter group alters text.

The following sections summarize the command set for **elvis**'s visual-command mode.

Visual-Mode Movement Commands

The following summarizes the visual mode's movement commands. *count* indicates that the command can be optionally prefaced by an argument that tells **elvis** how often to execute the command. *move* indicates that the command can be followed by a movement command, after which the command is executed on the text that lies between the point where the command was first typed and the point to which the cursor was moved. Typing the command a second time executes the command for the entire line upon which the cursor is positioned. *key* means that the command must be followed by an argument.

<ctrl-B>	Move up by one screenful.
[count] **<ctrl-D>**	Scroll down *count* lines (default, one-half screenful).
[count] **<ctrl-E>**	Scroll up *count* lines.
<ctrl-F>	Move down by one screenful.
<ctrl-G>	Show file status and the current line.
[count] **<ctrl-H>**	Move one character to the left.
[count] **<ctrl-J>**	Move down *count* lines.
<ctrl-L>	Redraw the screen.
[count] **<ctrl-M>**	Move to the beginning of the next line.
[count] **<ctrl-N>**	Move down *count* lines (default, one).
[count] **<ctrl-P>**	Move up *count* lines (default, one).
<ctrl-R>	Redraw the screen.
<ctrl-T>	Pop the tag stack — that is, return to the most recently tagged position. **elvis** removes that tag from the tag stack.
[count] **<ctrl-U>**	Scroll up *count* lines (default, one-half screenful).
[count] **<ctrl-X>**	Move the cursor to column *count* on the current line.
[count] **<ctrl-Y>**	Scroll down *count* lines.
<ctrl-]>	If the cursor is on a tag name, go to that tag.
<ctrl-^>	Switch to the previous file.
[count] **<space>**	Move right *count* spaces (default, one).
<quotation mark> *key*	Select which cut buffer to use next.
$	Move to the end of the current line.
%	Move to the matching (){}[] character.
[count] **%**	Move *count* percentage into the file. For example, the command **50%** moves the cursor to the middle of the file.
' *key*	Move to a marked line.
[count] **(**	Move backward *count* sentences (default, one).
[count] **)**	Move forward *count* sentences (default, one).
*****	Go to the next error in the error list.
[count] **+**	Move to the beginning of the next line.
[count] **,**	Repeat the previous **f** or **t** command, but move in the opposite direction.
[count] **-**	Move to the beginning of the preceding line.
[count] **.**	Repeat the previous *edit* command.
/ *text*	Search forward for *text,* which can be a regular expression.
0	If not part of a count, move to the first character of this line.
:	Switch to colon-command mode to execute one command.
[count] **;**	Repeat the previous **f** or **t** command.
? *text*	Search backwards for *text,* which can be a regular expression.
@ *key*	Execute the contents of a cut-buffer as **vi** commands.
[count] **B**	Move backwards *count* words (default, one).
[count] **E**	Move forwards *count* words (default, one).
[count] **F** *key*	Move left to the *count*'th occurrence of the given character (default, first).
[count] **G**	Move to to line *count* (default, last).
[count] **H**	Move to the top of the screen.
[count] **L**	Move to the bottom of the screen.
M	Move to the middle of the screen.
N	Repeat the last search, but in the opposite direction.
P	Paste text before the cursor.
Q	Shift to colon-command mode.
[count] **T** *key*	Move left *almost* to the given character.
U	Undo all recent changes to the current line.

V *[move...][command]*
Like **v**, described below, except it applies to whole lines. For example, the command **Vjj>** first highlights and then indents three lines. It is equivalent to **>2j** or **3>>**.

[count] **W** Move forward *count* words (default, one).
[count] **Y** Copy (or "yank") *count* lines into a cut buffer (default, one).
Z Z Save the file and exit.
[[Move back one section.
]] Move forward one section.
^ Move to the beginning of the current line, but after indent.
` *key* Move to the *key* character.
[count] **b** Move back *count* words.
[count] **e** Move forward to the end of the *count*'th word.
[count] **f** *c* Move rightward to the *count*'th occurrence of character *c*.
[count] **h** Move left *count* characters (default, one).
[count] **j** Move down *count* characters (default, one).
[count] **k** Move up *count* characters (default, one).
[count] **l** Move right *count* characters (default, one).
m *key* Mark a line or character.
n Repeat the previous search.
p Paste text after the cursor.
[count] **t** *key* Move rightward *almost* to the *count*'th occurrence of the given character (default, one).
u Undo the previous edit command.

v *[move ...][command]*
Highlight text as the cursor is moved, then apply *command* to the highlighted text. For example, **vwwwd** is approximately the same as **3dw**. To cancel the selection without altering the text, press **v** a second time.

[count] **w** Move forward *count* words (default, one).
y *move* Copy (or "yank") text into a cut buffer.
z *key* Scroll the screen, repositioning the current line as follows: **+** indicates top of the screen, **—** indicates the bottom, **.** indicates the middle.
[count] **{** Move back *count* paragraphs (default, one).
[count] **|** Move to the *count*'th column on the screen (leftmost, one).
[count] **}** Move forward *count* paragraphs (default, one).

If you are running **elvis** within an **X** terminal window, you can use also the mouse to reposition the cursor. To bypass this feature (e.g., to perform the standard X cut-and-paste tasks), press **<shift>** while clicking a mouse button.

Visual-Mode Edit Commands

The following describes the visual mode's editing commands.

! *[move]* Run the selected text through an external filter program.
!! Replace the current line with the output of an external command.
[count] **#** Increment a number by *count* (default, one).
[count] **&** Repeat the previous **:s//** command *count* times (default, once).
< *move* Shift the enclosed text left.
=*[move]* Filter the affected text. The default filter is **fmt**, which performs simple paragraph formatting and word wrap. To change the filter used by **=**, use the command **:set ep=***filter_name*.
> *move* Shift the enclosed text right.
[count] **A** *input* Append input to end of the line.
C *input* Change text from the cursor through the end of the line.
D Delete text from the cursor through the end of the line.
[count] **I** *input* Insert text at the beginning of the line (after indentations).
[count] **J** Join lines the current with the following line.
K Look up the word under the cursor. The default lookup program is **ref**. You can change **K** so as to get C language run-time library help for the word under the cursor by executing the command:

```
set kp="help -f/usr/lib/helpfile -i/usr/lib/helpindex -d@"
```

You can write this line into file **$HOME/.exrc**, which **elvis** reads before it begins execution.

[count] **O** *input*	Open a new line above the current line.
R *input*	Overtype.
[count] **S** *input*	Change lines, like **cc**.
[count] **X**	Delete *count* characters from the left of the cursor (default, one).
[count] **a** *input*	Insert text after the cursor.
c *move*	Change text.
d *move*	Delete text.
[count] **i** *input*	Insert text at the cursor.
[count] **o** *input*	Open a new line below the current line.
[count] **r** *key*	Replace *count* characters with text you type (default, one).
[count] **s** *input*	Replace *count* characters with text you type (default, one).
[count] **x**	Delete the character at which the cursor is positioned.
****	Pop up a menu of the most common operations.
[count] **~**	Toggle a character between upper case and lower case.

Colon-Mode Commands

The following summarizes the set of colon-mode commands. It is no accident that these commands closely resemble those for the **ed** line editor: they come, in fact, from **ex**, the editor upon which both **vi** (the UNIX visual editor) and **ed** derive. For that reason, colon-command mode is sometimes called **ex** mode.

line indicates whether the command can be executed on one or more lines. *line* can be a regular expression. Some commands can be used with an optional exclamation point; if done so, the editor assumes you know what you are doing and suppresses the warnings and prompts it would normally issue for these commands.

Most commands can be invoked simply by typing the first one or two letters of their names.

abbr *[word full_form]*
Define *word* as an abbreviation for *full_form*.

and
This command is used with the colon-mode command **if** to execute other commands conditionally. It is never used on its own. For more information, see the section *Conditional Commands* , below.

[line] **append**
Insert text after the current line.

args *[file1 ... fileN]*
With no arguments, print the files list on **elvis**'s command line. With one or more arguments, change the name of the current file.

cc *[files]*
Invoke the C compiler to compile *files*, and redirects all error messages into file **errlist**. After the compiler exits, scan the contents of **errlist** for error messages; if one is found, jump to the line and file indicated on the error line, and display the error message on the status line.

cd *[directory]*
Switch the current working directory. With no argument, switch to the **$HOME** directory.

[line][,line] **change** *["x]*
Replace the range of lines with the contents of cut-buffer *x*.

chdir *[directory]*
Same as the **cd** command.

color *[when] [[type] color] [on color]*
Set the screen's colors. This command works only if you have an ANSI-compatible color terminal. *when* defines the type of text whose color is being manipulated: **normal**, **standout**, **bold**, **underlined**, **italic**, **popup**, and **quit**. The default is normal. You may use the first letter of each as an abbreviation. *color* can be one of the following:

black	**blue**	**green**	**cyan**
red	**magenta**	**brown**	**white**
yellow	**gray**	**grey**	

Valid color *type*s can be one of the following: **light**, **bright**, or **blinking**.

The first use of **color** *must* specify both the foreground and background colors; the background color thereafter defaults to the background color of normal text. For example, the commands

```
color light cyan on blue
color b bright white
```

set the normal text to light cyan in the foreground and a blue background; and then set the foreground color for bold text to bright white.

Not every valid **color** command works as expected on the system console, due to limitations in the current release of the **ansipc** device driver.

[line][,line] **copy** *targetline*
Copy the range of lines to after the *targetline*.

[line][,line] **delete** *["x]*
Move the range of lines into cut buffer *x*.

digraph*[!] [XX [Y]]* Set *XX* as a digraph for *Y*. With no arguments, display all currently defined digraphs. With one argument, undefine the argument as a digraph.

edit*[!] [file]* Edit a file not named on the **elvis** command line.

else This command is used with the colon-mode command **if** to execute other commands conditionally. For more information, see the section *Conditional Commands* , below.

errlist*[!] [errlist]* Find the next error message in file **errlist**, as generated through **elvis**'s **cc** or **make** commands.

file *[file]* With an argument, change the output file to *file*. Without an argument, print information about the current output file.

[line][,line] **global** */regexp/ command*
Search the range of lines for all lines that contain the regular expression *regexp*, and execute *command* upon each.

if This command is used to execute other commands conditionally. For more information, see the section *Conditional Commands* , below.

[line] **insert** Insert text before the current line.

[line][,line] **join** Concatenate the range of lines into one line.

[line][,line] **list** Display the requested range of lines, making all embedded control characters explicit.

make *[target]* Same as the **cc** command, except that **make** is executed.

map*[!] key mapped_to*
Remap *key* to *mapped_to*. Normally, remapping applies just to visual-command mode; '!' tells **elvis** to remap the key under all modes. With no arguments, show all current key mappings.

[line] **mark** *x* Set a mark on *line*, and name it *x*.

mkexrc Save current configuration into file **./.exrc**, which will be read next time you invoke **elvis**.

[line][,line] **move** *targetline*
Move the range of lines to after *targetline*.

next*[!] [files]* Switch to the next file on the **elvis** command line.

Next*[!]* Switch to the preceeding file on the **elvis** command line.

[line][,line] **number** Display the range of lines, with line numbers.

or This command is used with the colon-mode command **if** to execute other commands conditionally. For more information, see the section *Conditional Commands* , below.

pop Pop the tag stack — that is, return to the most recently tagged position. **elvis** removes that tag from the tag stack.

previous*[!]* Switch to the preceeding file on the **elvis** command line.

[line][,line] **print** Display the specified range of lines.

[line] **put** *["x]* Copy text from cut buffer *x* after the current line.

quit*[!]* Quit **elvis**, and return to the shell.

[line] **read** *file* Read the contents of *file* and insert them after **line** (default, the last line).

rewind*[!]* Switch to the first file on the **elvis** command line.

[line[,*line]]***s/***oldstring***/***newstring***/***[***g***]*
Substitute the first instance of *newstring* for *oldstring*. If no range of lines is indicated, the substitution is performed only on the current line. To change every instance of *oldline* into *newline* on a line, append the suffix **g** ("global") to this command.

The command **s** with no arguments repeats the previous substitution. It is a synonym for the command **&**, described below.

set *[options]* Set an **elvis** option. For details, see the section on *set Options*, below

shell Invoke a shell.

source *file* Read a set of colon-mode commands from *file*, and execute them.

[line][,line] **substitute** */regexp/replacement/[p][g][c]*
For the range of lines, replace the first instance of *regexp* with *replacement*. *p* tells **elvis** to print the *last* line upon which a substitution was performed. *g* means perform a global substitution, i.e., replace all instances of *regexp* on each line with *replacement*. *c* tells **elvis** to ask for confirmation before performing each substitution.

tag*[!] tagname* Find *tagname* in file **tags**, which records information about all tags. If found, jump to the file and line upon which the tag is set.

then This command is used with the colon-mode command **if** to execute other commands conditionally. For more information, see the section *Conditional Commands* , below.

[line][,line] **to** *targetline*
Copy the range of lines to after the *targetline*.

unabbr *word* Unabbreviate *word*.

undo Undo the last editing command.

unmap*[!] key* Unmap *key*.

version Display the current version of **elvis**.

[line][,line] **vglobal** */regexp/ command*
Search the range of lines for all lines that do not contain the regular expression *regexp*, and execute *command* upon each.

visual Enter visual-command mode.

wq Save the changed file, and exit.

[line][,line] **write***[!] [[>>][file]*
Write the file being edited into *file*. With the >> argument, append the edited text onto the end of *file*.

xit*[!]* Same as the **wq** command, described above, except that it does not write files that have not changed.

[line][,line] **yank** *["x]* Copy the range of lines into cut buffer *x*.

[line][,line] **!** *command*
Execute *command* under a subshell, then return.

[line][,line] **<** Shift the range of lines left by one tabwidth.

[line][,line] **=** With no range of lines specified, print the number of the current line. With line arguments, print the endpoints of the lines in question, and the number of lines that lie between them. (Remember, *line* can be a regular expression as well as a number.)

[line][,line] **>** Shift the range of lines right by one tabwidth.

[line][,line] **&** Repeat the last substitution command.

@ x Read the contents of cut-buffer *x* as a set of colon-mode commands, and execute them. With no arguments, list all current settings.

\@ Beginning with release 1.8, **elvis** replaces the escape sequence **\@** by the word that the cursor is on. This works in two special contexts: in regular expressions, and in any **ex** command that also replaces '%' with the current file name. This escape sequence can simplify writing certain kinds of macros.

Conditional Commands

Beginning with release 1.8pl3, **elvis** supports conditional commands. Some of these commands set a conditional-execution flag; others examine that flag and perform (or do not perform) commands if the flag is set. You cannot nest conditional commands.

The colon-mode commands **if**, **and**, and **or** test for a condition. Their syntax is typical: each must be followed by an expression, and **and** and **or** must follow an initial **if** command. Each command tests for a single condition. That condition may involve examining the options set by the command **set** (which are described in detail in the next section), set by **termcap** values, or by constants.

Colon-mode commands **then** and **else** execute commands conditionally, based upon the value of the conditional-execution flag set by a preceding **if** command.

These commands most often are embedded in an initialization file, to initialize **elvis** properly under a variety of conditions. The following gives an example **if** command that can be embedded in a user's **.exrc** file. The command correctly sets up the colors for both the system console and for an X terminal window. It works around the fact that the console can handle color, but an X terminal window cannot:

```
if term="console"
then color yellow on blue | color quit white on blue
else color black on white
```

To disable these commands, add **-DNO_IF** to **CFLAGS**, then recompile **elvis**.

set Options

As noted above, the command **set** can set **elvis**'s internal options. Options come in three flavors: *boolean*, which turn on or off a feature of the editor; *string* which define the string associated with a particular action; (e.g., the name of a command or feature); and *numeric*, which set a dimension for the editor (e.g., the number of rows or columns on the terminal screen). To turn off a boolean option, prefix it with the string "no".

The following lists the options that **set** recognizes. Assume that the boolean options are on, unless the entry says otherwise:

autoindent (boolean)	Auto-indent during input? Default is **no**.
autoprint (boolean)	When in **ex** mode, print the current line.
autotab (boolean)	Can auto-indent use tabs?
autowrite (boolean)	Is auto-write on when switching files? Default is **no**.
beautify (boolean)	Should the editor strip control characters from a file? Default is **no**.
charattr (boolean)	Interpret \fX sequences? Default is **no**.
cc (string)	Name of the C compiler. Default is **cc -c**.
columns (numeric)	Width of the screen. Default is 80.
digraph (boolean)	Recognize digraphs? Default is **no**.
directory (string)	Where are temporary files kept? Default is **/usr/tmp**.
edcompatible (boolean)	Remember ":s//" options? Default is **no**.
equalprg (boolean)	Program to run for the '=' operator. Default is **fmt**.
errorbells (boolean)	Beep when an error occurs.
exrc (boolean)	Read the **./.exrc** file? Default is **no**.
exrefresh (boolean)	Write lines individually when in **ex** mode.
flash (boolean)	Use visible alternative to bell.
flipcase (string)	Non-ASCII chars flipped by the tilde character '~'. Default is the NULL string.
hideformat (boolean)	Hide text formatter commands.
ignorecase (boolean)	Make searches case sensitive. While in **ignorecase** mode, the searching mechanism does not distinguish between an upper-case letter and its lower-case form. In **noignorecase** mode, upper case and lower case are treated as being different. Default is **no**.
inputmode (boolean)	Start **vi** in insert mode? Default is **no**.
keytime (numeric)	Timeout for mapped key entry. Default is two.
keywordprg (string)	Path name of program invoked by **shift-K**. Default is **ref**.
lines (numeric)	Number of lines on the screen. Default is 25.
list (boolean)	Display lines in **list** mode? Default is **no**.
magic (boolean)	Enable the use of regular expressions in a search. While in **magic** mode, all meta-characters behave as described above. In **nomagic** mode, only **^** and **$** retain their special meaning.
make (string)	Name of the "make" program. Default is **make**.
mesg (boolean)	Allow messages from other users?
modelines (boolean)	Are mode lines processed? Default is **no**.
more (boolean)	Pause between messages?
nearscroll (numeric)	This governs when to scroll versus when to redraw the screen. If you move the cursor more than the number of lines set by this option, **elvis** redraws the screen; otherwise, it scrolls the screen. The default is 15 lines.
novice (boolean)	Set options for ease of use? Default is **no**.
number (boolean)	Turn on line numbering.
paragraphs (string)	Names of **nroff** "paragraph" commands. Default is **PPppIPLPQP**.
prompt (boolean)	Show ':' prompt in **ex** mode.
readonly (boolean)	Prevent overwriting of original file. Default is **no**.
remap (boolean)	Allow key maps to call other key maps.
report (numeric)	Report when a given number of changes occur. Default is five.
ruler (boolean)	Display line and column numbers. Default is **no**.
safer	Toggle **elvis'** security option. This option is set temporarily during the execution of any command from modeline or **./.exrc**. It disables the following commands:

:!	**:Next**	**:abbreviate**	**:args**	**:cc**
:cd	**:chdir**	**:ex**	**:file**	**:make**
:map	**:mkexrc**	**:next**	**:pop**	**:previous**
:rewind	**:shell**	**:stop**	**:suspend**	**:tag**
:unab	**:unmap**	**:visual**	**:write**	

Note that **set safer** does not disable **:wq**, as this command does not let the user name the file into which to write. **:read** is still allowed, but it will not let the user read from a filter.

set safer forbids the user from altering the following options:

autowrite	**cc**	**directory**	**equalprg**
keywordprg	**make**	**shell**	**trapunsafe**

	It also disables wildcard expansion and the visual '!' command.
scroll (numeric)	Set the number of lines the screen scrolls with the **<ctrl-D>** and **<ctrl-U>** commands. Default is 12.
sections (string)	Names of **nroff** "section" commands. Default is **NHSHSSSEse**.
shell (string)	Path name of the shell. Default is **/bin/sh**.
showmatch (boolean)	Show all matching parentheses, brackets, and braces. Default is **no**.
showmode (boolean)	Say when editor is in input mode. Default is **no**.
shiftwidth (numeric)	Set number of characters the **<** and **>** commands shift the screen. Default is eight.
sidescroll (numeric)	Set number of columns the editor scrolls. Default is eight.
sync (boolean)	Call **sync()** often? Default is **no**.
tabstop (numeric)	Number of columns set by a tab character. Default is eight.
taglength (numeric)	Number of significant characters in a tag name. Default is zero.
tags (string)	Name the list of "tags" files that **elvis** can read.
tagstack (boolean)	Enable the tagstack. Default is **no**.
term (string)	Name of the current terminal's **termcap** entry. Default is **$TERM**.
terse (boolean)	Give shorter error messages? Default is **no**.
timeout (boolean)	Distinguish **<esc>** from an arrow key?
warn (boolean)	Warn if a file has been modified?
window (numeric)	Number of lines to redraw after long move. Default is 24.
wrapmargin (numeric)	Left margin to use when wrapping long lines in input mode. Default is zero.
wrapscan (boolean)	Searches wrap from end to beginning of the file.
writeany (boolean)	Let the write command **:w** clobber a file. Default is **no**.

Input-Mode Commands

Most keystrokes are interpreted as being text and inserted directly into the text; however, some keystrokes are still interpreted as commands. Thus, you can perform an entire session of simple editing directly within input mode without switching to either of the command modes.

The following summarizes the commands that can be executed directly within input mode:

<ctrl-A>	Insert a copy of the last input text.
<ctrl-C>	Send the signal **SIGINT** to interrupt a command.
<ctrl-D>	Delete one indent character.
<ctrl-H>	Erase the character before the cursor.
<ctrl-L>	Redraw the screen.
<ctrl-M>	Insert a newline.
<ctrl-P>	Insert the contents of the cut buffer.
<ctrl-R>	Redraw the screen, like **<ctrl-L>**.
<ctrl-T>	Insert an indent character.
<ctrl-U>	Move to the beginning of the line. When you are typing a command line or search pattern on the bottom line, **<ctrl-U>** backspaces over all characters typed so far.
<ctrl-V>	Insert the following keystroke, even if special.
<ctrl-W>	Backspace to the beginning of the current word.
<ctrl-Z><ctrl-Z>	Write the file and exit **elvis**.
<ctrl-Z>	Save the file if it has been modified, but do not exit from **elvis**. This works only if you have set the mode **autowrite**.
<esc>	Shift from input mode to visual-command mode.
<del>	Delete the current character.

When **elvis** is in input mode, you can use the keystroke **<ctrl-O>** to invoke *some* visual commands without exiting from input mode. For example, when you are in input mode, typing **<ctrl-O>J** moves down a line but leaves you in input mode.

Keyboard Macros

elvis Beginning with release 1.8, **elvis** can record keystokes into a cut buffer. This is equivalent to a MicroEMACS "keyboard macro".

The following commands manipulate keyboard macros:

[a Open a keyboard macro. **elvis** executes all subsequent keystrokes as normal, but also records them within a temporary buffer.

]a Stop recording keystrokes, and copy the keystrokes into the cut buffer.

@a To replay the recorded keystrokes.

Command-line Options

elvis lets you name up to 27 files on the command line, thus allowing you to edit up to 27 files simultaneously. The "next file" and "previous file" commands described above allow you to shift from one file to another during the same editing session; in this way, for example, you can cut text from one file and paste it into another.

elvis recognizes the following command-line options:

-r Recover a previous edit.
-R Invoke **elvis** in "read-only" mode. This is equivalent to invoking **elvis** via the link **view**.
-s Invoke **elvis** in "safer" mode. This is equivalent to the command **set safer**, described above.
-t *tag* Begin editing at *tag*.
-m *[file]*
Invoke **elvis** in error-handling mode. It searches through *file* for something that looks like an error message from a compiler, then positions the cursor at that point for editing.
-e Begin in colon-command mode.
-v Begin in visual-command mode.
-i Begin in input mode.
-w *winsize*
Set the value of option **window**, which sets the size of the screen with which **elvis** works, to *winsize*. **window** is described below.
+*command*
Execute *command* immediately upon beginning editing. For example

```
elvis +237 foo
```

causes **elvis** to move directly to line 237 immediately upon beginning to edit file **foo**.

Regular Expressions

elvis uses regular expressions for searching and substitutions. A regular expression is a text string in which some characters have special meanings. This is much more powerful than simple text matching.

elvis's **regexp** package treats the following one- or two-character strings (called meta-characters) in special ways:

\(\) Delimit subexpressions. When the regular expression matches a chunk of text, **elvis** remembers which portion of that chunk matched the subexpression. The command

```
:s/regexp/newtext/
```

command makes use of this feature.

^ Match the beginning of a line. For example, to find **foo** at the beginning of a line, use the regular expression **/^foo/**. Note that **^** is a metacharacter only if it occurs at the beginning of a regular expression; anywhere else, it is treated as a normal character.

$** Match the end of a line. It is a metacharacter only when it occurs at the end of a regular expression; elsewhere, it is treated as a normal character. For example, the expression **/$$/ searches for a dollar sign at the end of a line.

\< Match a zero-length string at the beginning of a word. A word is a string of one or more letters and digits; it can begin at the beginning of a line or after one or more non-alphanumeric characters.

\> Matches a zero-length string at the end of a word. A word can end at the end of the line or before one or more non-alphanumeric characters. For example, **/\<end\>/** finds any instance of the word **end**, but ignores any instances of "end" inside another word, such as "calendar".

. Match any single character.

[*character-list***]**
Match any single character from the *character-list*. Inside the *character-list*, you can denote a span of characters by writing the first and last characters, with a hyphen between them. If the character-list is preceded by a **^**, then the list is inverted — it matches all characters not mentioned in the list. For example, **/[a-zA-Z]/** matches any letter, and **/[^]/** matches anything other than a blank.

\{*n***\}** Repeat the preceding expression *n* times. This operator can only be placed after something that matches a single character. For example, **/^-\{80\}$/** matches a line of eighty hyphens, and **/\<[a-zA-Z]\{4\}\>/** matches any four-letter word.

\{*n***,***m***\}** Repeat the preceding single-character expression between *n* and *m* times, inclusive. If the *m* is omitted (but the comma is present) then it is taken to be infinity. For example, **/"[^"]\{3,5\}"/** matches any pair of quotation marks that enclose three, four, or five non-quotation characters.

* Repeat the preceding single-character expression zero or more times. For example, **/.*/** matches a whole line.

/+ Repeat the preceding single-character expression one or more times. It is equivalent to **\{1,\}**. For example, **/.\+/** matches a whole line, but only if the line contains at least one character. It does not match empty lines.

/? The preceding single-character expression is optional — that is, that it can occur zero or one times. It is equivalent to **\{0,1\}**. For example, **/no[-]?one/** matches **no one**, **no-one**, and **noone**.

Anything else is treated as a normal character that must exactly match a character from the scanned text. The special strings may all be preceded by a backslash to force them to be treated normally.

Substitutions

The command **:s** has at least two arguments: a regular expression and a substitution string. The text that matches the regular expression is replaced by text that is derived from the substitution string.

Most characters in the substitution string are copied into the text literally but a few have special meaning:

& Insert a copy of the original text.
~ Insert a copy of the previous replacement text.
\1 Insert a copy of that portion of the original text that matched the first set of parentheses.
\2-**\9** Do the same for the second and all subsequent pairs of parentheses.
\U Convert all characters of any later **&** or **\#** to upper case.
\L Convert all characters of any later **&** or **\#** to lower case.
\E End the effect of **\U** or **\L**.
\u Convert the first character of the next **&** or **#** to upper case.
\l Convert the first character of the next **&** or **\#** to lower case.

These may be preceded by a backslash to force them to be treated normally.

If **nomagic** mode is in effect, then **&** and **~** will be treated normally, and you must write them as **\&** and **\~** for them to have special meaning.

Preserving Text

Should **elvis** sense that it is about to die unexpectedly, it invokes the command **elvprsv** to save the temporary file in which it manipulates the file you are editing. To recover this saved file, use the command **elvrec**. Both commands are described in the Lexicon.

Initialization Files

When you invoke **elvis**, it searches for file **$HOME/.exrc**. If it finds that file, it reads the file and attempts to execute its contents as a series of **ex** commands. (As noted earlier, **ex** commands simply are **elvis'** set of colon-mode commands, but without the preceding colon.)

Usually, this file is used to contain instances of the commands **set** and **color**, to set up **elvis'** environment and appearance to your taste. For example, if your **.exrc** file contains the commands

```
color white on blue
set ignorecase
set inputmode
```

then **elvis** sets the screen's background color to blue and its foreground color to white; turn on **ignorecase** mode (that is, string searches will ignore case), and come up in input mode rather than command mode.

The file **$HOME/elvis.rc** is a synonym for **$HOME/.exrc**.

When you invoke **elvis**, it also searches for the file **$HOME/.exfilerc** This file holds **ex** commands that **elvis** executes every time it loads a text file for editing. You can embed **if** commands in this file so that **elvis** handles special classes of files uniquely. For example, you can use an **if** command to tell **elvis** to handle files with the suffix **.c** differently from other files; this lets you invoke special editing functions for C programs.

Examples

The first example changes every occurrence of "utilize" to "use":

```
:%s/utilize/use/g
```

The next example deletes all white space that occurs at the end of a line anywhere in the file. (The brackets contain a single space and a single tab character):

```
:%s/[   ]+$//
```

The next example converts the current line to upper case:

```
:s/.*/U&/
```

The next example underlines each letter in the current line, by changing it into an **underscore backspace letter** sequence. (The **<ctrl-H>** is entered as **<ctrl-V><backspace>**):

```
:s/[a-zA-Z]/_^H&/g
```

The last example locates the last colon in a line, and swaps the text before the colon with the text after the colon. The first pair of parentheses delimits the stuff before the colon, and the second pair delimits the stuff after. In the substitution text, **\1** and **\2** are given in reverse order to perform the swap:

```
:s/\(.*\):\(.*\)/\2:\1/
```

Environment

elvis reads the following environmental variables:

TERM This names your terminal's entry in the **termcap** or **terminfo** data base.

TERMCAP
Optional. If your system uses **termcap**, and the **TERMCAP** variable is not set, then **elvis** reads your terminal's definition from **/etc/termcap**. If **TERMCAP** is set to the full path name of a file (beginning with a '/'), it reads your terminal's description from the named file instead of from **/etc/termcap**. If **TERMCAP** is set to a value that does not begin with a '/', then **elvis** assumes that its value is the full **termcap** entry for your terminal.

TERMINFO
Optional. **elvis** treats this exactly like the environmental variable **TERMCAP**, except for the **terminfo** data base.

LINES
COLUMNS
Optional. These variables, if set, override the screen-size values given in the **termcap** or **terminfo** description of your terminal On windowing systems such as X, **elvis** has other ways to determine the screen size, so you should probably leave these variables unset.

EXINIT
Optional. This variable can hold **ex** commands that **elvis** executes before it reads any **.exrc** files.

SHELL Optional. This variable sets the default value for the **shell** option, which determines which shell program **elvis** uses to perform wildcard expansion in file names, and to execute filters or external programs. The default value is **/bin/sh**.

HOME This variable should be set to the name of your home directory. **elvis** looks for its initialization file there. If **HOME** is not set, then **elvis** does not execute the initialization file.

TAGPATH
Optional. This variable is used by the program **ref**. See "ref" for more information.

Bug Fixes from Release 1.7

Beginning with release 4.2.10, COHERENT includes **elvis** release 1.8pl3. The following describes the bugs that this release fixes The initial release of **elvis** 1.8 includes the following bug fixes:

- Most screen update bugs are fixed. Most of ones that were not fixed can be avoided by **:set nooptimize**.
- A bug in the visual '@' command was fixed. This bug can be blamed for most of **elvis**' incompatibility with fancy macro packages. **elvis** can now run the "Bouncing Ball," "Word Completion," and "Turing" macros with no changes. NB, it still cannot run "Towers of Hanoi."

The following bug fixes are included in patch-level 1 (**pl1**):

- Fixed a bug that caused core dump when you use the '}' command used on blank line after last paragraph in file.
- Fixed a bug that caused loss of text with **AutoIndent** enabled, when two newlines are inserted into the middle of a line.

The following bug fixes are included in patch-level 2 (**pl2**):

- Fixed a security hole on some UNIX systems.
- After **:w**, **#** refers to the file just written.
- Fixed bug in tag lookup.
- The compiler error parser now allows '_' in a file name.
- Fixed a bug that caused some blank lines in the file **.exrc** to be interpreted as **:p** commands.
- Increased the limit on word size for the command **<ctrl-A>**. The old limit was 30; the new limit is 50. If you exceed this limit, **elvis** will now search for the longest possible substring; before, it would bomb. To change the limit, add **-DWSRCH_MAX=***n* (where *n* gives the limit on word size) to **CFLAGS** in the **Makefile**, then recompile **elvis**.
- Increased the size of an array used while showing option settings. The old size could overflow if you did a **:set all** on some systems. Now, the maximum size is calculated at compile time, and the array is declared to this size.
- The command **5r<ctrl-M>** now leaves the cursor in the right place. In earlier releases, **5r<ctrl-M>** would replace five characters with five newline characters, and leave the cursor five lines lower. Release 1.8 replaced five characters with a single newline character, to mimic the real **vi** better, but still left the cursor five lines lower. This patch finally makes it right.

The following bug fixes are included in patch-level 2 (**pl2**):

- Corrected bugs in **:tag** and **:make**, which caused tag addresses and error messages to be forgotten after switching files. The **.exfilerc** feature interacted with these bugs, and made them pretty obnoxious. A similar bug caused the command **:e +cmd file** to start misbehaving; it has been fixed, too.
- The option **window** now defaults to zero. Zero is a special value, which means "use as many rows as possible." Previously, this option defaulted to the maximum number of rows available when **elvis** started (usually 24), which resulted in '@' signs appearing on the screen if you resized the display while **elvis** was running. This problem only showed up when you ran **elvis** in an X terminal window.
- A bug has been fixed in autoindentation. Previously, if you inserted a newline before the first non-whitespace character on a line, then everything after the insertion point was wiped out. (This is different from the bug that **pl2** fixed. **pl2**'s fix addresses a bug that affected insertion of multiple newlines anywhere in a lines; this one affects inserting a single newline before the first non-whitespace character.)
- To avoid linking problems on various systems, the variable **kD** has been renamed **kDel**, and function **ioctl()** in **pc.c** renamed **elvis_ioctl**.
- A bug that caused **:!** to clobber the value of **#** (i.e., the previous file name) has been fixed.
- There is a bug that affects screen redraws after pasting (the visual **p** and **P** commands). In an attempt to work around this bug, **elvis** will sometimes redraw the screen from scratch after a multi-line paste.
- Some people have reported problems using **fmt** on non-English text. I suspect that this is due to a faulty implementation of **isspace()** in the standard C library. In release 1.8pl3, **fmt** does not use **isspace()** anymore; it uses a built-in macro which explicitly tests for **<space>** or **<tab>**. This may solve the problem.

Files

/tmp/elv* — Temporary files
tags — Data base used by the **tags** command
$HOME/.exrc — File that sets personal defaults
$HOME/.exfilerc — File that sets defaults when a file is read
$HOME/elvis.rc — Same as **.exrc**

See Also

commands, ed, elvprsv, elvrec, ex, fmt, me, vi, view

Notes

elvis returns zero if the file being edited was updated. It returns one if the file was not updated, and a different nonzero value if an error occurred.

Full documentation for **elvis** is included with this release in compressed file **/usr/src/alien/Elvis.doc.Z**.

elvis is copyright © 1990 by Steve Kirkendall, and was written by Steve Kirkendall (kirkenda@cs.pdx.edu or uunet!tektronix!psueea!eecs!kirkenda), assisted by numerous volunteers. It is freely redistributable, subject to the restrictions noted in included documentation. Source code for **elvis** is available through the Mark Williams bulletin board, USENET, and numerous other outlets.

elvprsv — Command

Preserve the modified version of a file after a crash
elvprsv ["-*why elvis died***"] /tmp/***filename***...**
elvprsv -R /tmp/*filename***...**

The command **elvprsv**, or "elvis preserved," preserves your edited text should **elvis** die unexpectedly. You can later use the command **elvrec** to rebuild the edited buffer.

You should never need to run **elvprsv** from the command line. **elvis** automatically invokes it should it sense that it is about to die. Script **/etc/rc** should also invoke **elvprsv**, to preserve any temporary files that may have been left in directory **/tmp** when the system went down.

If **elvis** were to die unexpectedly while you were editing a file, **elvprsv** would preserve the most recent version of your text. The preserved text is stored in a special directory; **elvprsv** does *not* overwrite your text file. **elvprsv** sends mail to each user whose work it preserves. Should the preservation directory not be set up correctly, **elvprsv** simply sends you a mail message that describes how to it manually.

Files

/tmp/elv*
    Temporary file that **elvis** was using when it died.
/usr/preserve/p*
    Text that is preserved by **elvprsv**.
/usr/preserve/Index
    Text file that names all preserved files and the files in which they are preserved.

See Also

commands, elvis, elvrec

Notes

Due to the permissions on directory **/usr/preserve**, only the superuser **root** can run **elvprsv**.

If you were editing a nameless buffer when **elvis** died, **elvprsv** saves its contents in a file named **foo**.

elvprsv was written by Steve Kirkendall (kirkenda@cs.pdx.edu).

elvrec — Command

Recover the modified version of a file after a crash
elvrec [*preservedfile* **[***newfile***]]**

Should **elvis** die while you were editing a file, it automatically invokes the command **elvprsv** to preserve the most recent version of your edited text. **elvprsv** stores the preserved text in a special directory: it does *not* overwrite your text file

The command **elvrec** locates the preserved version of a file, and either overwrites your text file or creates a new file, whichever you prefer. The recovered file will hold nearly all of your changes.

To see a list of all recoverable files, run **elvrec** with no argument. *preservedfile* names the file into which **elvprsv** had saved the edited buffer. **elvrec** is very picky about file names: you must use exactly the same path name as you did to edit the file.

newfile names the file into which **elvrec** writes the edited buffer. If you do not name a *newfile* on its command line, **elvrec** overwrites your original file with the preserved, edited version.

Files

/usr/preserve/p*
The text that was preserved when **elvis** died.
/usr/preserve/Index
The names of all preserved files, and the names of the files that preserve their text.

See Also

commands, elvis, elvprsv

Notes

Due to the permissions on the directory **/usr/preserve**, only the superuser **root** can run **elvrec**.

If you haven't set up a directory for file preservation, then you must manually run the program **elvprsv** instead of **elvrec**.

If you were editing a nameless buffer when **elvis** died, then **elvrec** saves the text into a file named **foo**.

elvrec was written by Steve Kirkendall (kirkenda@cs.pdx.edu).

em87 — Kernel Module

Perform/emulate hardware floating-point operations

The kernel module **em87** performs or emulates hardware floating-point operations. Whether it performs the operations or emulates them depends whether your computer contains a mathematics co-processor. Note that the Intel 80486-DX processor has the co-processor built in.

em87 is called a *kernel module* because you can link it into the kernel or exclude it from the kernel, just like a device driver. However, it is not a true device driver because it does not perform I/O from a peripheral device. To install **em87** into a kernel (should your kernel not already contain it), log in as the superuser **root** and execute the following commands:

```
cd /etc/conf
em87/mkdev
bin/idmkcoh -o /kernel_name
```

where *kernel_name* is the name of the new kernel to build. When you next boot COHERENT, hardware floating point will be enabled.

See Also

device drivers, float, kernel

emacs — Command

COHERENT screen editor
emacs [-e *errorfile***] [-f** *bindfile***] [***textfile ...***]**

emacs is a link to the COHERENT screen editor, which is a scaled-down version of the EMACS screen editor.

For details, see the Lexicon entry for **me**.

See Also

commands, me

enable — Command

Enable a port
/etc/enable *port...*

The COHERENT system is a multiuser operating system; it allows many users to use the system simultaneously. An asynchronous communication *port* connects each user to the system, normally by a terminal or a modem attached to the port. The system communicates with the port by means of a character special file in directory **/dev**, such as **/dev/com3r** or **/dev/com2l**.

The COHERENT system will not allow a user to log in on a port until the system creates a *login process* for the port. The **enable** command tells the system to create a login process for each given *port*. For example, the command

```
/etc/enable com1r
```

enables port **/dev/com1r**.

enable changes the entry for each given *port* in the terminal characteristics file **/etc/ttys**. The baud rate specified in **/etc/ttys** must be the appropriate baud rate for the terminal or modem connected to the port. See the Lexicon entry for **ttys** for more information.

The command **disable** disables a port. The command **ttystat** checks whether a port is enabled or disabled.

Files

/etc/ttys — Terminal characteristics file
/dev/com* — Devices serial ports

See Also

asy, commands, disable, getty, login, ttys, ttystat

Diagnostics

enable normally returns one if it enables the *port* successfully and zero if not. If more than one *port* is specified, **enable** returns the success or failure status of the last port it finds. It returns -1 if it cannot find any given *port*. An exit status of -2 indicates an error.

Notes

It is not recommended that you attempt to enable a port that is already enabled. To make sure, run **/etc/disable** before running **/etc/enable**.

endgrent() — General Function (libc)

Close group file
#include <grp.h>
endgrent()

endgrent() closes the file **/etc/group**. It returns NULL if an error occurs.

Files

/etc/group
<grp.h>

See Also

group, libc

endhostent() — Sockets Function (libsocket)

Close file /etc/hosts
#include <netdb.h>
void endhostent();

The function **endhostent()** is one of a set of functions that interrogate the file **/etc/hosts** to look up information about a remote host on a network. It closes **/etc/hosts** upon the conclusion of searching.

See Also

gethostbyaddr(), gethostbyname(), libsocket, sethostent()

endnetent() — Sockets Function (libsocket)

Close network file
#include <netdb.h>
void endnetent();

Function **endnetent()** closes file **/etc/networks** which describes all entities on your local network, after it had been opened by function **getnetent()** or **setnetent()**.

See Also

getnetbyaddr(), getnetent(), libsocket, netdb.h, setnetent()

endprotoent() — Sockets Function (libsocket)

Close protocols file
#include <netdb.h>
void endprotoent();

Function **endprotoent()** closes file **/etc/protocols** which describes all protocols recognized by your local network, after it had been opened by function **getprotoent()** or **setprotoent()**.

See Also

getprotobyaddr(), getprotobyname(), getprotoent(), libsocket, netdb.h, setprotoent()

endpwent() — General Function (libc)

Close password file
#include <pwd.h>
endpwent()

The COHERENT system has five routines that search the file **/etc/passwd**, which contains information about every user of the system. **endpwent()** closes the password file. Please note that this function does not return a meaningful value.

Example

For an example of this function, see the entry for **getpwent()**.

Files

/etc/passwd
pwd.h

See Also

getpwent(), getpwnam(), getpwuid(), libc, pwd.h, setpwent()

endservent() — Sockets Function (libsocket)

Close protocols file
#include <netdb.h>
void endservent();

Function **endservent()** closes file **/etc/protocols** which describes the services offered by TCP/IP on your local network. after it had been opened by function **getservent()** or **setservent()**.

See Also

getservbyname(), getservbyport(), getservent(), libsocket, netdb.h, setservent()

endspent() — General Function (libc)

Close the shadow-password file
#include <shadow.h>
endspent()

The COHERENT system has four routines that search the file **/etc/shadow**, which contains the password of every user of your system. **endspent()** closes **/etc/shadow**. It does not return a meaningful value.

Files

/etc/shadow
/usr/include/shadow.h

See Also

getspent(), libc, setspent(), shadow, shadow.h

endutent() — General Function (libc)

Close the login logging file
#include <utmp.h>
void endutent()

Function **endutent()** closes the logging file. Usually this is the system file **/etc/utmp**. The file must have been opened by a call to function **getutent()**, **getutid()**, or **getutline()**.

For a summary of the family of functions that manipulate logging files, see the Lexicon entry for **utmp.h**.

See Also

getutent(), libc, utmp.h

enum — C Keyword

Declare a type and identifiers

An **enum** declaration is a data type whose syntax resembles those of the **struct** and **union** declarations. It lets you enumerate the legal value for a given variable. For example,

```
enum opinion {yes, maybe, no} GUESS;
```

declares type **opinion** can have one of three values: **yes**, **no**, and **maybe**. It also declares the variable **GUESS** to be of type **opinion**.

As with a **struct** or **union** declaration, the tag (**opinion** in this example) is optional; if present, it may be used in subsequent declarations. For example, the statement

```
register enum opinion *op;
```

declares a register pointer to an object of type **opinion**.

All enumerated identifiers must be distinct from all other identifiers in the program. The identifiers act as constants and can be used wherever constants are appropriate.

COHERENT assigns values to the identifiers from left to right, normally beginning with zero and increasing by one. In the above example, the values of **yes**, **no**, and **maybe** are set, respectively, to one, two, and three. The values often are **int**s, although if the range of values is small enough, the **enum** will be an **unsigned char**. If an identifier in the declaration is followed by an equal sign and a constant, the identifier is assigned the given value, and subsequent values increase by one from that value; for example,

```
enum opinion {yes=50, no, maybe} guess;
```

sets the values of the identifiers **yes**, **no**, and **maybe** to 50, 51, and 52, respectively.

See Also

C keywords
ANSI Standard, §6.5.2.2

ENV — Environmental Variable

File read to set environment

Whenever the Korn shell is invoked, it executes the script named in the environmental variable **ENV**. By custom, this is set to **${HOME}/.kshrc**, although you can name any file you wish. This script usually sets aliases and environmental variables, and executes the **set** command to modify the behavior of the shell itself.

By defining **ENV** in your **.profile**, you can ensure that this file is executed whenever you invoke a shell. If you wish to have its definitions read only by the login shell, insert the instruction

```
unset ENV
```

at the end of the script named by **ENV**.

See Also

environmental variables, ksh, .kshrc

env — Command

Execute a command in an environment
env [-] [*VARIABLE***=***value ...***] [***command args***]**

The command **env** executes *command* with *args*, modifying the existing environment by performing the requested assignments.

The '-' option tells **env** to replace the environment with the arguments of the form **VARIABLE=***value*; otherwise the assignments are added to the environment.

If *command* is omitted, the resulting environment is printed.

See Also

commands

environ — C Language

Process environment

extern char **environ;

environ is an array of strings, called the *environment* of a process. By convention, each string has the form

name=value

Normally, each process inherits the environment of its parent process. The shell **sh** and various forms of **exec** can change the environment. The shell adds the name and value of each shell variable marked for *export* to the environment of subsequent commands. The shell adds assignments given on the same line as a command to the environment of the command, without affecting subsequent commands.

See Also

C language, exec, getenv(), Programming COHERENT, putenv(), sh
POSIX Standard, §3.1.2

environmental variables — Technical Information

The *environment* is a set of information that is read by all programs that run on your system. It consists of one or more *environmental variables* that you set. For example, when you set the environmental variable **PATH**, you tell COHERENT that you wish to pass this information to all programs on your system, including COHERENT itself.

By changing the environment, you can change the way a command works without rewriting any commands that you may have embedded in batch files, scripts, or **makefile**s.

Your programs may request environmental variables of their own definition. COHERENT uses the following environmental variables to set its environment. Note that the variables marked with an asterisk are used only by the Korn shell **ksh**.

ASKCC . . . . Have **mail** prompt for CC names
CWD* . . . . Current working directory
EDITOR . . . . Editor used by default by **mail**
ENV* . . . . File read to set environment
FCEDIT* . . . . Editor used by the **fc** command
IFS . . . . Characters recognized as white space
HOME . . . . User's home directory
KSH_VERSION*. . . . . List current version of Korn shell
LASTERROR* . . . . . . Program that last generated an error
LIBPATH. . . . . Directories that hold compiler phases and libraries
LOGNAME . . . . Name user's identifier
MAIL . . . . File that holds user's mail messages
MLP_COPIES . . . . . . Set default number of copies to print
MLP_FORMLEN. . . . . Set default page length
MLP_LIFE . . . . Set default life for print jobs
MLP_PRIORITY. . . . . Set default priority for print spooling
MLP_SPOOL. . . . . . . Pass user-specific information to print spooler
PAGER . . . . User's preferred output filter
PATH . . . . Directories that hold executable files
PS1 . . . . User's default prompt
PS2 . . . . Prompt when unbalanced quotation marks span a line
SECONDS*. . . . . Number of seconds since current shell started
SHELL . . . . Name the default shell
TERM. . . . . Name the default terminal type
TIMEZONE. . . . . User's current time zone
TMPDIR . . . . Directory that holds temporary files

USER . . . . Name user's identifier

You can also set the following environmental variables to control the default settings of the COHERENT assembler **as**, the C compiler **cc**, and the linker **ld**:

ARHEAD . . . . Append options to beginning of **ar** command line
ARTAIL. . . . . Append options to end of **ar** command line
ASHEAD . . . . Append options to beginning of **as** command line
ASTAIL . . . . Append options to end of **as** command line
CCHEAD . . . . Append options to beginning of **cc** command line
CCTAIL. . . . . Append options to end of **cc** command line
CPPHEAD . . . . Append options to beginning of **cpp** command line
CPPTAIL . . . . Append options to end of **cpp** command line
LDHEAD . . . . Append options to beginning of **ld** command line
LDTAIL. . . . . Append options to end of **ld** command line

See Also

get_env(), unset, Using COHERENT

Notes

To delete an environmental variable, use the command **unset**.

envp — C Language

Argument passed to main()
char *envp[];

envp is an abbreviation for environmental parameter. It is the traditional name for a pointer to an array of string pointers passed to a C program's **main** function, and is by convention the third argument passed to **main**.

Example

The following example demonstrates **envp**, **argc**, and **argv**.

```
#include <stdio.h>

main(argc, argv, envp)
int argc;                        /* Number of args */
char *argv[];                    /* Argument ptr array */
char *envp[];                    /* Environment ptr array */
{
      int a;

      printf("The command name (argv[0]) is %s\n", argv[0]);
      printf("There are %d arguments:\n", argc-1);
      for (a=1; a<argc; a++)
            printf("\targument %2d:\t%s\n", a, argv[a]);

      printf("The environment is as follows:\n");
      a = 0;
      while (envp[a] != NULL)
            printf("\t%s\n", envp[a++]);
}
```

See Also

argc, argv, C language, environ, main()

EOF — Manifest Constant

Indicate end of a file
#include <stdio.h>

EOF is an indicator that is returned by several STDIO functions to indicate that the current file position is the end of the file.

Many STDIO functions, when they read **EOF**, set the end-of-file indicator that is associated with the stream being read. Before more data can be read from the stream, its end-of-file indicator must be cleared. Resetting the file-position indicator with the functions **fseek**, **fsetpos**, or **ftell** will clear the indicator, as will returning a character to the stream with the function **ungetc**.

See Also

file, manifest constant, stream, stdio.h
ANSI Standard, §7.9.1

epson — Command

Prepare files for Epson printer
epson [-cdfnrw8] [-b *head* **] [-i** *n* **] [-o** *file* **] [-s** *n* **] [** *file ...* **]**

epson prepares text for printing an Epson or Epson-compatible dot-matrix printer. It recognizes the **nroff** output sequences for boldface and italics and converts them into the Epson codes for emphasized print and italics.

If you do not name a file on its command line, **epson** reads the standard input. By default, **epson** writes its output to the standard output. Thus, you can use **epson** as a filter within an MLP backend script.

By default, **epson** outputs the string "\033 @ \033 t \0" at the beginning of each job to initialize the printer. The sequence "\033 @" clears the printer and prepares it to receive new data; while the escape sequence "\033 t \0" makes an Epson printer's built-in italics font available. To suppress the italics-font portion of the initialization sequence, use the command-line option **-n**, described below.

epson recognizes the following command-line options:

-b *head* Print the given *head* as a double-width banner at the top of the first output page.

-c Use compressed printing mode.

-d Print boldface as double strikes. Normally, **epson** recognizes the sequence "*c***\b***c*" as boldface and prints *c* in emphasized printing mode. **-d** is useful in conjunction with **-c**.

-f Do not print a formfeed character at the end of each *file*.

-i*n* Indent *n* spaces at the start of each output line.

-n No italics: suppress the italics portion of the printer-initialization string. When you use this switch, **epson** outputs the string "\033 @" to initialize the printer.

-o *file* Write output into *file*, instead of sending it to device **/dev/lp**.

-r Print all characters in Roman; do not use italics. Normally, **epson** recognizes the sequence "_**\b***c*" as italic and prints *c* in its italic character set.

-s*n* Print *n* newlines at the end of each line. *n* must be 1, 2, or 3; the default is 1.

-w Use double width printing mode.

-8 Print lines with vertical spacing of eight lines per inch instead of the default six lines per inch.

See Also

commands, lp, nroff, pr, printer

Notes

Prior to release 4.2.12 of COHERENT, **epson** wrote its output to device **/dev/lp** instead of to the standard output.

erand48() — Random-Number Function (libc)

Return a 48-bit pseudo-random number as a double
double erand48(*xsubi***)**
unsigned short *xsubi***[3];**

Function **erand48()** generates a 48-bit pseudo-random number, and returns it in the form of a **double**. The value returned is (or should be) uniformly distributed through the range of 0.0 through 1.0. *xsubi* is an array of three unsigned short integers from which the pseudo-random number is built.

See Also

libc, srand48()

errno — Global Variable

External integer for return of error status
#include <errno.h>
extern int errno;

errno is an external integer that COHERENT links into each of its programs. COHERENT sets **errno** to the negative value of any error status returned to any function that performs COHERENT system calls.

Mathematical functions use **errno** to indicate classifications of errors on return. **errno** is defined within the header file **errno.h**. Because not every function uses **errno**, it should be polled only in connection with those functions that document its use and the meaning of the various status values. For the names of the error codes (as defined in **errno.h**, their value, and the message returned by the function **perror**, see **errno.h**.

Example

For an example of using **errno** in a mathematics program, see the entry for **acos**.

See Also

errno.h, libm, perror(), Programming COHERENT, signal()
ANSI Standard, §7.1.4
POSIX Standard, §2.4

errno.h — Header File

Error numbers used by errno()
#include <errno.h>

errno.h is the header file that defines and describes the error numbers returned in the external variable **errno**. The following lists the error numbers defined in **errno.h**:

EPERM: Permission denied
You lack permission to perform the operation you have requested.

ENOENT: No such file or directory
A program could not find a required file or directory.

ESRCH: No such process
You are attempting to communicate with a process that does not exist.

EINTR: Interrupted system call
A COHERENT system call failed because it received a signal or an alarm expired.

EIO: I/O error
A physical I/O error occurred on a device driver. This could be a tape error, a CRC error on a disk, or a framing error on a synchronous HDLC link.

ENXIO: No such device or address
You attempted to access a device that does not exist. It may be that a specified minor device is invalid, or the unit is powered off. This error can also indicate that a block number given to a minor device is out of range. If you attempt to open a pipe in write-only mode, if **O_NDELAY** is set, and if there are currently no readers on this pipe, **open()** returns immediately and sets **errno** to **ENXIO**.

E2BIG: Argument list too long
The number of bytes of arguments passed in an **exec** is too large.

ENOEXEC: exec() format error
The file given to **exec** is not a valid executable module (probably because it does not have the magic number at the beginning), even though its mode indicates that it is executable.

EBADF: Bad file descriptor
You passed a file descriptor to a system call for a file that was not open or was opened in a manner inappropriate to the call. For example, a file descriptor opened only for reading may not be accessed for writing.

ECHILD: No child processes
A process issued a **wait()** call when it had no outstanding children.

EAGAIN: No more processes
The system cannot create any more processes, either because it is out of table space or because the invoking process has reached its quota or processes.

ENOMEM: not enough memory
The system does not have enough memory available to map a process into memory. This occurs in response to a the system calls **exec()** or **brk()**.

EACCES: Permission denied
You do not have permission to perform the requested operation upon a given file.

EFAULT: Bad address
You requested an address that does not lie within the address space. Normally, this generates signal **SIGSYS**, which terminates the process.

ENOTBLK: Block device required
You passed to system calls **mount()** and **umount()** the descriptor of file that is not a block-special device.

EBUSY: Mount device busy
You passed to the system call **mount()** the file descriptor of a device that is already mounted; or you passed to the system call **umount()** the descriptor of a device that has open files or active working directories.

EEXIST: File exists
An attempt was made to **link** to a file that already exists.

EXDEV: Cross-device link
You attempted to link a file on one file system with a file on another. This is not permitted.

ENODEV: No such device
You attempted to manipulate a device that does not exist.

ENOTDIR: Not a directory
You attempted to perform a directory operation upon a file that is not a directory. For example, you passed the file descriptor of a character-special device to system calls **chdir()** or **chroot()**.

EISDIR: Is a directory
You attempted to perform an inappropriate operation upon a directory. For example, you passed the file descriptor of a directory to **write()**.

EINVAL: Invalid argument
An argument to a system call is out of range. For example, you passed to **kill()** or **umount()** the file descriptor of a device that is not mounted.

ENFILE: File table overflow
The COHERENT kernel uses a static table to record which files are open. This error indicates that this table is full. Until a file is closed, thus freeing space on this table, no more files can be opened on your system.

EMFILE: Too many open files
The COHERENT kernel limits the number of files that any one process can have open at any given time; this error indicates that you have exceeded this number. The system call **sysconf()** returns the number of files that a process can open (among other items of information). For details, see its entry in the Lexicon.

ENOTTY: Not a teletypewriter (tty)
You attempted to perform a terminal-specific operation upon a device which is not a terminal.

ETXTBSY: Text file busy
The text segment of a shared load module is unwritable. Therefore, an attempt to execute it while it is being written or an attempt to open it for writing while it is being executed will fail.

EFBIG: File too large
The block-mapping algorithm for a file fails above 1,082,201,088 bytes. Attempting to write a file larger than this will generate this error.

ENOSPC: No space left on device
You attempt to write onto a device that is full. If the attemped write was onto a file system, either the file system's supply of blocks was exhausted, or its supply of i-nodes was exhausted.

ESPIPE: Tried to seek on a pipe
It is illegal to invoke the system call **lseek()** on a pipe.

EROFS: Read-only file system
You attempted to write onto a file system mounted read-only.

EMLINK: Too many links
A file can have no more than 32,767 links. The attempted link operation would exceed this value.

EPIPE: Broken pipe
You attempted to invoke the system call **write()** on a pipe for which there are no readers. This condition is accompanied by the signal **SIGPIPE,** so the error will be seen only if the signal is ignored or caught.

EDOM: Mathematics library domain error
An argument to a mathematical routine falls outside that function's domain.

ERANGE: Mathematics library result too large
The result of a mathematical function is too large to be represented.

ENOMSG: No message of desired type
You invoked **msgrcv()** to read a message of a given type, but none was waiting to be read.

EIDRM: Identifier removed

EDEADLK: Deadlock condition
A process is deadlocked for some reason.

ENOLCK: No record locks available
The maximum number of record locks has been exceeded.

ENOSTR: Device not a stream
You attempted to perform a STREAMS operation on a file that is not a stream.

ENODATA: No data available

ETIME: Timer expired

ENOSR: Out of STREAMS resources

ENOPKG: Package not installed

EPROTO: Protocol error

EBADMSG: Not a data message

ENAMETOOLONG: File name too long

EOVERFLOW: Value too large for defined data type

ENOTUNIQ: Name not unique on network

EBADFD: File descriptor in bad state

EREMCHG: Remote address changed

ELIBACC: Cannot access a needed shared library
COHERENT does not yet support shared libraries.

ELIBBAD: Accessing a corrupted shared library
COHERENT does not yet support shared libraries.

ELIBSCN: **.lib** section in **a.out** corrupted

ELIBMAX: Maximum number of shared libraries exceeded
COHERENT does not yet support shared libraries.

ELIBEXEC: Cannot **exec()** a shared library directly
COHERENT does not yet support shared libraries.

EILSEQ: Illegal byte sequence

ENOSYS: Operation not applicable

ELOOP: Symbolic links error.
Number of symbolic links encountered during path name traversal exceeds **MAXSYMLINKS**. COHERENT does not yet support symbolic links.

EUSERS: Too many users

ENOTSOCK: Socket operation on non-socket

EDESTADDRREQ: Destination address required

EMSGSIZE: Message too long

EPROTOTYPE: Protocol wrong type for socket

ENOPROTOOPT: Protocol not available

EPROTONOSUPPORT: Protocol not supported

ESOCKTNOSUPPORT: Socket type not supported

EOPNOTSUPP: Operation not supported on transport endpoint

EPFNOSUPPORT: Protocol family not supported

EAFNOSUPPORT: Address family not supported by protocol family

EADDRINUSE: Address already in use

EADDRNOTAVAIL: Cannot assign requested address

ENETDOWN: Network is down

ENETUNREACH: Network is unreachable

ENETRESET: Network dropped connection because of reset

ECONNABORTED: Software-caused connection abort

ECONNRESET: Connection reset by peer

ENOBUFS: No buffer space available

EISCONN: Transport endpoint is already connected

ENOTCONN: Transport endpoint is not connected

ESHUTDOWN: Cannot send after transport endpoint shutdown

ETIMEDOUT: Connection timed out

ECONNREFUSED: Connection refused

EHOSTDOWN: Host is down

EHOSTUNREACH: No route to host

EALREADY: Operation already in progress

EINPROGRESS: Operation now in progress

ESTALE: Stale NFS file handle
COHERENT does not yet support nonproprietary file systems.

See Also

errno, header files, perror(), signal()
ANSI Standard, §7.1.3
POSIX Standard, §2.4

eval — Command

Evaluate arguments
eval [*token ...*]

The shell normally evaluates each token of an input line before executing it. During evaluation, the shell performs parameter, command, and file-name pattern substitution. The shell does *not* interpret special characters after performing substitution.

eval is useful when an additional level of evaluation is required. It evaluates its arguments and treats the result as shell input. For example,

```
A='>file'
echo a b c $A
```

simply prints the output

```
a b c >file
```

because '>' has no special meaning after substitution, but

```
A='>file'
eval echo a b c $A
```

redirects the output

```
a b c
```

to **file**. Similarly,

```
A='$B'
B='string'
echo $A
eval echo $A
```

prints

```
$B
string
```

In the first **echo** the shell performs substitution only once.

The shell executes **eval** directly.

See Also

commands, ksh, sh

ex — Command

Berkeley-style line editor
ex [*options* **] [+***cmd* **] [** *file1 ... file27* **]**

ex is a link to **elvis**, which is a clone of the UNIX **vi/ex** set of editors. Invoking **elvis** through this link forces it to operate solely in colon-command mode, just as the UNIX **ex** editor operates.

For information on how to use this version of **ex**, see the Lexicon page for **elvis**.

See Also

commands, ed, elvis, me, vi, view

Notes

elvis is copyright © 1990 by Steve Kirkendall, and was written by Steve Kirkendall (kirkenda@cs.pdx.edu or ...uunet!tektronix!psueea!eecs!kirkenda), assisted by numerous volunteers. It is freely redistributable, subject to the restrictions noted in included documentation. Source code for **elvis** is available through the Mark Williams bulletin board, USENET, and numerous other outlets.

Please note that **elvis** is distributed as a service to COHERENT customers, as is. It is not supported by Mark Williams Company. *Caveat utilitor.*

exec — Command

Execute command directly
exec [*command*]

The shell normally executes commands through the system call **fork()**, which creates a new process. The shell command **exec** directly executes the given *command* through one of the **exec()** functions instead. Normally, this terminates execution of the current shell.

If the *command* consists only of redirection specifications, **exec** redirects the input or output of the current shell accordingly without terminating it. If the *command* is omitted, **exec** has no effect.

See Also

commands, execution, fork(), ksh, sh, xargs
POSIX Standard, §3.1.2

execl() — General Function (libc) (libc)

Execute a load module
#include <unistd.h>
execl(*file, arg0, arg1, ..., argn,* **NULL**)
char **file, *arg0, *arg1, ..., *argn***;**

The function **execl()** calls the COHERENT system call **execve()** to execute a program. It specifies arguments individually, as a NULL-terminated list of *arg* parameters. For more information on file execution, see **execution**.

See Also

execution, execve(), getuid(), libc, unistd.h
POSIX Standard, §3.1.2

Diagnostics

execl() does not return if successful. It returns -1 for errors, such as *file* being nonexistent, not accessible with execute permission, having a bad format, or too large to fit in memory.

execle() — General Function (libc) (libc)

Execute a load module
#include <unistd.h>
execle(*file, arg0, arg1, ..., argn,* **NULL,** *env*)
char **file, *arg0, *arg1, ..., *argn***, char** **env***[];**

The function **execle()** calls the COHERENT system call **execve()** to execute a program. It first initializes the new stack of the process to contain a list of strings that are command arguments. It specifies arguments individually, as a NULL-terminated list of *arg* parameters. The argument *envp* points to an array of pointers to strings that define *file*'s environment. For more information on program execution and environments, see **execution**.

See Also

environ, execution, execve(), libc, unistd.h
POSIX Standard, §3.1.2

Diagnostics

execle() does not return if successful. It returns -1 for errors, such as *file* being nonexistent, not accessible with execute permission, having a bad format, or being too large to fit into memory.

execlp() — General Function (libc)

Execute a load module
#include <unistd.h>
execlp(*file, arg0, arg1, ..., argn,* **NULL)**
char **file, *arg0, *arg1, ..., *argn***;**

The function **execlp()** calls the COHERENT system call **execve()** to execute a program. It initializes the new stack of the process to contain a list of strings that are command arguments. It specifies arguments individually, as a NULL-terminated list of *arg* parameters. Unlike the related function **execl()**, **execlp()** searches for *file* in all directories named in the environmental variable **PATH**. For more information on program execution, see **execution**.

See Also

environ, execution, execve(), libc, unistd.h
POSIX Standard, §3.1.2

Diagnostics

execlp() does not return if successful. It returns -1 for errors, such as *file* not existing in the directories named in **PATH**, not accessible with execute permission, having a bad format, or too large to fit in memory.

execlpe() — General Function (libc)

Execute a load module
execlpe(*file, arg0, arg1, ..., argn,* **NULL,** *env***)**
char **file, *arg0, *arg1, ..., *argn***;**
char **env***[];**

The function **execlpe()** calls the COHERENT system call **execve()** to execute a program. It initializes the new stack of the process to contain a list of strings that are command arguments. It specifies arguments individually, as a NULL-terminated list of *arg* parameters.

The argument *env* points to an array of pointers to strings that define *file*'s environment.

Unlike the related function **execl()**, **execlpe()** searches for *file* in all directories named in the environmental variable **PATH**— that is, the current **PATH**, not the one contained within the environmented pointed to by *env*.

For more information on program execution, see **execution**.

See Also

environ, execution, execl(), execvep(), libc

Diagnostics

execlpe() does not return if successful. It returns -1 for errors, such as *file* not existing in the directories named in **PATH**, not accessible with execute permission, having a bad format, or too large to fit in memory.

execlpe() is not part of the SVID specification. Therefore, it may not be present on non-COHERENT operating systems.

execution — Definition

Program execution under COHERENT is governed by the various forms of the COHERENT system call **exec()**. This call allows a process to execute another executable *file* (or load module). This is described in **coff.h**.

The code, data and stack of *file* replace those of the requesting process. The new stack contains the command arguments and its environment, in the format given below. Execution starts at the entry point of *file*.

During a successful call to **exec()**, the system deactivates profiling, and resets any caught signals to **SIG_DFL.**

Every process has a real-user id, an effective-user id, a saved-effective user id; and a real-group id, an effective-group id, and a saved-effective group id. These identifiers are defined in the Lexicon entries for, respectively, **setuid()** and **setgid()**. For most load modules, **exec()** does not change any of these. However, if the *file* is marked with the set user id or set group id bit (see **stat()**), **exec()** sets the effective-user id (effective-group id) of the process to the user id (group id) of the *file* owner. In effect, this changes the file access privilege level from that of the real id to that of the effective id. The owner of *file* should be careful to limit its abilities, to avoid compromising file security.

exec() initializes the new stack of the process to contain a list of strings, which are command arguments. **execl()**, **execle()**, **execlp()**, and **execlpe()** specify arguments individually, as a NULL-terminated list of *arg* parameters. **execv()**, **execve()**, **execvp()**, and **execvpe()** specify arguments as a single NULL-terminated array **argv** of parameters.

The **main()** function of a C program is invoked in the following way:

```
main(argc, argv, envp)
int argc;
char *argv[], *envp[];
```

argc is the number of command arguments passed through **exec()**, and **argv** is an array of the actual argument strings. **envp** is an array of strings that comprise the process environment. By convention, these strings are of the form *variable=value*, as described in the Lexicon entry **environ**. Typically, each *variable* is an **export**ed shell variable with the given *value*.

execl() and **execv()** simply pass the old environment, referenced by the external pointer **environ**.

execle(), **execlpe()**, **execve()**, and **execvpe()** pass a new environment *env* explicitly.

execlp(), **execlpe()**, **execvp()**, and **execvpe()** search for *file* in each of the directories indicated by the shell variable **$PATH**, in the same way that the shell searches for a command. These calls execute a shell command *file*. Note that **execlpe()** and **execvpe()** search the current **PATH**, not the **PATH** contained within the environment pointed to by *env*.

Files

/bin/sh — To execute command files

See Also

environ, exec(), execl(), execle(), execlp(), execlpe(), execv(), execve(), execvp(), execvpe(), fork(), ioctl(), Programming COHERENT, signal(), stat(), xargs

Diagnostics

None of the **exec()** routines returns if successful. Each returns -1 for an error, such as if *file* does not exist, is not accessible with execute permission, has a bad format, or is too large to fit in memory.

Notes

Each **exec()** routine now examines the beginning of an executable file for the token **#!**. If found, it invokes the program named on that line and passes it the rest of the file. For example, if you wish to ensure that a given script is executed by the by the Bourne shell **/bin/sh**, begin the script with the line:

```
#!/bin/sh
```

execv() — General Function (libc)

Execute a load module
#include <unistd.h>
execv(*file, argv***)**
char **file*, **argv***[];**

The function **execv()** calls the COHERENT system call **execve()** to execute a program. It specifies arguments as a single, NULL-terminated array of parameters, called *argv*. **execv()** passes the environment of the calling program to the called program. For more information on program execution, see **execution**.

See Also

environ, execution, execve(), libc, unistd.h
POSIX Standard, §3.1.2

Diagnostics

execv() does not return if successful. It returns -1 for errors, such as *file* being nonexistent, not accessible with execute permission, having a bad format, or too large to fit in memory.

execve() — System Call (libc)

Execute a load module
#include <unistd.h>
execve(*file, argv, env***)**
char **file*, **argv***[],** **env***[];**

The function **execve()** executes a program. It specifies arguments as a single, NULL-terminated array of parameters, called *argv*. The argument *env* is the address of an array of pointers to strings that define *file*'s environment. This allows **execve()** to pass a new environment to the program being executed. For more information on program execution, see **execution**.

Example

The following example demonstrates **execve()**, as well as **tmpnam()**, **getenv()**, and **path()**. It finds all lines with more than **LIMIT** characters and calls MicroEMACS to edit them.

```
#include <stdio.h>
#include <path.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>

#define LIMIT 70

extern **environ, *tempnam();

main(argc, argv)
int argc; char *argv[];
{
      /*                 me     -e   tmp   file */
      char *cmda[5] = { NULL, "-e", NULL, NULL, NULL };
      FILE *ifp, *tmp;
      char line[256];
      int  ct, len;

      if ((NULL == (cmda[3] = argv[1])) ||
          (NULL == (ifp = fopen(argv[1], "r")))) {
            fprintf(stderr, "Cannot open %s\n", argv[1]);
            exit(EXIT_FAILURE);
      }

      if ((cmda[0] = path(getenv("PATH"), "me", X_OK)) == NULL) {
            fprintf(stderr, "Cannot locate me\n");
            exit(EXIT_FAILURE);
      }

      if (NULL == (tmp = fopen((cmda[2] = tempnam(NULL, "lng")), "w"))) {
            fprintf(stderr, "Cannot open tmpfile\n");
            exit(EXIT_FAILURE);
      }

      for (ct = 1; NULL != fgets(line, sizeof(line), ifp); ct++)
            if (((len = strlen(line)) > LIMIT) ||
                ('\n' != line[len -1]))
                  fprintf(tmp, "%d: %d characters long\n", ct, len);

      fclose(tmp);
      fclose(ifp);

      if (execve(cmda[0], cmda, environ) < 0) {
            fprintf(stderr, "cannot execute me\n");
            exit(EXIT_FAILURE);
      }
}
```

See Also

environ, execution, libc, unistd.h
POSIX Standard, §3.1.2

Diagnostics

execve() does not return if successful. It returns -1 for errors, such as *file* being nonexistent, not accessible with execute permission, having a bad format, or too large to fit in memory.

execvp() — General Function (libc)

Execute a load module
#include <unistd.h>
execvp(*file, argv***)**
char ****file*, **argv*[];**

The function **execvp()** calls the COHERENT system call **execve()** to execute a program. It specifies arguments as a single, NULL-terminated array of parameters, called *argv*. Unlike the related call **execv()**, **execvp()** searches for *file* in all of the directories named in the environmental variable **PATH**. For more information on program execution, see **execution**.

See Also

environ, execution, execve(), libc, unistd.h
POSIX Standard, §3.1.2

Diagnostics

execvp() does not return if successful. It returns -1 for errors, such as *file* being nonexistent, not accessible with execute permission, having a bad format, or too large to fit in memory.

execvpe() — General Function (libc)

Execute a load module
execvp(*file, argv***,** *env***)**
char ****file,* ****argv*[],** ****env***[];**

The function **execvpe()** calls the COHERENT system call **execve()** to execute a program. It specifies arguments as a single, NULL-terminated array of parameters, called *argv*. The argument *env* is the address of an array of pointers to strings that define *file*'s environment. This allows **execvpe()** to pass a new environment to the program being executed.

Unlike the related call **execv()**, **execvpe()** searches for *file* in all of the directories named in the environmental variable **PATH**— that is, the current **PATH**, not the one contained within the environmented pointed to by *env*.

For more information on program execution, see **execution**.

See Also

environ, execution, execv(), execve(), libc

Diagnostics

execvp() does not return if successful. It returns -1 for errors, such as *file* being nonexistent, not accessible with execute permission, having a bad format, or too large to fit in memory.

execvpe() is not part of the SVID specification. Therefore, it may not be present on non-COHERENT operating systems.

exit — Command

Exit from a shell
exit [*status***]**

exit terminates a shell. If the optional *status* is specified, the shell returns it; otherwise, the previous status is unchanged. From an interactive shell, **exit** sets the *status* if specified, but does not terminate the shell. The shell executes **exit** directly.

See Also

commands, ksh, sh

exit() — General Function (libc)

Terminate a program gracefully
#include <stdlib.h>
void exit(*status***) int** *status***;**

The library function **exit()** is the normal method to terminate a program directly. *status* information is passed to the parent process. By convention, an exit status of zero indicates success, whereas an exit status greater than zero indicates failure. If the parent process issued a **wait()** call, it is notified of the termination and is passed the least significant eight bits of *status*. As **exit()** never returns, it is always successful. Unlike the system call **_exit()**, **exit()** does extra cleanup, such as flushing buffered files and closing open files.

Example

For an example of this function, see the entry for **fopen()**.

See Also

_exit(), atexit(), close(), EXIT_FAILURE, EXIT_SUCCESS, libc, stdlib.h, wait()
ANSI Standard, §7.10.4.3
POSIX Standard, §8.1

Notes

If you do not explicitly set *status* to a value, the program returns whatever value happens to have been in the register EAX. You can set *status* to either **EXIT_SUCCESS** or **EXIT_FAILURE**.

EXIT_FAILURE — Manifest Constant

Indicate program failed to execute successfully
#include <stdlib.h>

EXIT_FAILURE is a manifest constant that is defined in the header **stdlib.h**. It is used as an argument to the function **exit()** to indicate that the program failed to execute successfully.

See Also

exit(), manifest constant, stdlib.h
ANSI Standard, §7.10.4.3

EXIT_SUCCESS — Manifest Constant

Indicate program executed successfully
#include <stdlib.h>

EXIT_SUCCESS is a manifest constant that is defined in the header **stdlib.h**. It is used as an argument to the function **exit()**, to indicate that the program executed successfully.

See Also

exit(), manifest constant, stdlib.h
ANSI Standard, §7.10.4.3

exp() — Mathematics Function (libm)

Compute exponent
#include <math.h>
double exp(*z*) double *z*;

exp() returns the exponential of *z*, or *e*^*z*.

Example

The following example, called **apr.c**, computes the annual percentage rate (APR) for a given rate of interest. Compile it with the command:

```
cc -f apr.c -lm
```

It is by Brent Seidel (brent_seidel@chthone.stat.com):

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
	double rate, APR;
	char buffer[50];

	printf("Enter interest rate in percent (e.g., 12.9): ");
	fflush(stdout);

	if (gets(buffer) == NULL)
		exit(EXIT_FAILURE);
	rate = strtod(buffer);

	APR = (exp(rate/100.0) - 1) * 100.0;
	printf("The APR for %g%% compounded daily is %g%%\n", rate, APR);
}
```

See Also

errno, frexp(), ldexp(), libm
ANSI Standard, §7.5.4.1
POSIX Standard, §8.1

Diagnostics

exp() indicates overflow by an **errno** of **ERANGE** and a huge returned value.

export — Command

Add a shell variable to the environment
export [*name* ...]
export [*name*=*value*]

When the shell executes a command, it passes the command an *environment*. By convention, the environment consists of assignments, each of the form *name*=*value*. For example, typing

```
export TERM=vt100
```

sets the environmental variable **TERM** to equal the string **vt100**.

A command may look for information in the environment or may simply ignore it. In the above example, a program that reads the variable **TERM** (such as COHERENT) will assume that you are working on a DEC VT-100 terminal or one that emulates it.

The shell places the *name* and the value of each shell variable that appears in an **export** command into the environment of subsequently executed commands. It does not place a shell variable into the environment until it appears in an **export** command.

With no arguments, **export** prints the name and the value of each shell variable currently marked for export.

The shell executes **export** directly.

See Also

commands, environ, exec, ksh, sh

expr — Command

Compute a command-line expression
expr *argument ...*

The arguments to **expr** form an expression. **expr** evaluates the expression and writes the result on the standard output. Among other uses, **expr** lets the user perform arithmetic in shell command files.

Each *argument* is a separate token in the expression. An argument has a logical value 'false' if it is a null string or has numerical value zero, 'true' otherwise. Integer arguments consist of an optional sign followed by a string of decimal digits. The range of valid integers is that of signed long integers. No check is made for overflow or illegal arithmetic operations. Floating point numbers are not supported.

The following list gives each **expr** operator and its meaning. The list is in order of increasing operator precedence; operators of the same precedence are grouped together. All operators associate left to right except the unary operators '!', '-', and '**len**', which associate right to left. The spaces shown are significant - they separate the tokens of the expression.

{ *expr1***,** *expr2***,** *expr3* **}**
Return *expr2* if *expr1* is logically true, and *expr3* otherwise. Alternatively, **{** *expr1* **,** *expr2* **}** is equivalent to **{** *expr1* **,** *expr2* **, 0 }**.

expr1 **|** *expr2*
Return *expr1* if it is true, *expr2* otherwise.

expr1 **&** *expr2*
Return *expr1* if both are true, zero otherwise.

expr1 relation expr2
Where *relation* is one of <, <=, >, >=, ==, or !=, return one if the *relation* is true, zero otherwise. The comparison is numeric if both arguments can be interpreted as numbers, lexicographic otherwise. The lexicographic comparison is the same as **strcmp** (see **string**).

expr1 **+** *expr2*

expr1 **-** *expr2*
Add or subtract the integer arguments. The expression is invalid if either *expr* is not a number.

expr1 * *expr2*

expr1 **/** *expr2*

expr1 **%** *expr2*
: Multiply, divide, or take remainder of the arguments. The expression is invalid if either *expr* is not numeric.

expr1 **:** *expr2*
: Match patterns (regular expressions). *expr2* specifies a pattern in the syntax used by **ed**. It is compared to *expr1*, which may be any string. If the \(...\) pattern occurs in the regular expression the matching operator returns the matched field from the string; if there is more than one \(...\) pattern the extracted fields are concatenated in the result. Otherwise, the matching operator returns the number of characters matched.

len *expr*
: Return the length of *expr*. It behaves like **strlen** (see **string**). *len* is a reserved word in **expr**.

!*expr* Perform logical negation: return zero if *expr* is true, one otherwise.

-*expr* Unary minus: return the negative of its integer argument. If the argument is non-numeric the expression is invalid.

(*expr* **)**
: Return the *expr*. The parentheses allow grouping expressions in any desired way.

The following operators have special meanings to the shell **sh**, and must be quoted to be interpreted correctly: **{ } ( ) < > & |** *.

See Also

commands, ed, ksh, sh, test

Notes

expr returns zero if the expression is true, one if false, and two if an error occurs. In the latter case an error message is also printed.

extern — C Keyword

Declare storage class

extern indicates that a C element belongs to the *external* storage class. Both variables and functions may be declared to be **extern**. Use of this keyword tells the C compiler that the variable or function is defined outside of the present file of source code. All functions and variables defined outside of functions are implicitly **extern** unless declared **static**.

When a source file references data that are defined in another file, it must declare the data to be **extern**, or the linker will return an error message of the form:

```
undefined symbol name
```

For example, the following declares the array **tzname**:

```
extern char tzname[2][32];
```

When a function calls a function that is defined in another source file or in a library, it should declare the function to be **extern**. In the absence of a declaration, **extern** functions are assumed to return **int**s, which may cause serious problems if the function actually returns a 32-bit pointer (such as on the 68000 or i8086 LARGE model), a **long**, or a **double**.

For example, the function **malloc** appears in a library and returns a pointer; therefore, it should be declared as follows:

```
extern char *malloc();
```

If you do not do so, the compiler assumes that **malloc** returns an **int**, and generate the error message

```
integer pointer pun
```

when you attempt to use **malloc** in your program.

See Also

auto, C keywords, pun, register, static, storage class
ANSI Standard, §6.5.1