4. EX COMMAND MODE

Ex is an editing mode in which elvis acts like a line editor. This means that you type in a command line, and when the line is complete elvis executes it on the current text buffer. I.e., in ex each *line* (or group of lines) is a command, as opposed to vi where each *character* (or group of characters) is a command.

Typically, ex commands are used to do perform complex actions such as global search & replace, or actions which require an argument such as writing the edit buffer out to a different file.

Ex is also used as the configuration language for elvis; configuration scripts such as elvis.ini, .exrc (or elvis.rc), and elvis.arf contain a series of ex commands.

You can switch freely between vi and ex. If you're in vi mode, you can enter a single ex command line via the visual : command, or more permanently switch via the visual Q command. If you're in ex mode, you can switch to vi mode via ex's :vi command.

Normally elvis will start in vi mode, but you can force it to start in ex mode by supplying a **-e** command line flag. On UNIX systems, you can link elvis to a name which ends with "x" to achieve the same effect.

The remainder of this section discusses how to enter lines, the general syntax of an ex command line, and the specific commands which elvis supports.

4.1 Entering lines

In elvis, when you're typing in an ex command line you're really inputing text into a buffer named "Elvis ex history". All of the usual input mode commands are available, including **Backspace** to erase the previous character, **Control-W** to erase the previous word, and so on.

Any previously entered lines will still be in the "Elvis ex history" buffer, and you can use the arrow keys to move back and edit earlier commands. You can even use the **Control-O** input-mode command with the ?*regexp* visual command, to search for an earlier command line.

When you hit the **Enter** key on a line in the "Elvis ex history" buffer, elvis sends that line to the ex command parser, which is described in the next section.

4.1.1 An example

Suppose you enter the command...

```
:e ~/proj1/src/header.h
```

...and then realize that you really wanted "header2.h" instead of "header.h". You simplest way to get "header2.h" is to...

1) Hit the : key to start a new ex command line.

2) Hit the **Up** arrow key, or **^O k** to move back to the preceding command line (which was ":e ~/proj1/src/header.h"). **^O k** works because **^O** reads and executes one vi command, and the **k** vi command moves the cursor back one line. The **Up** arrow key works because it is mapped to "visual k", which does exactly the same thing as **^O k**.
3) Hit the **Left** arrow key twice, or **^O 2 h**, to move the cursor back to the '.' character in "header.h".
4) Hit **2** to insert a '2' before the '.' character. At this point, the line should look like ":e ~/proj1/src/header2.h".
5) Hit **Enter** to submit the revised command line.

Or suppose you really wanted "footer2.h" instead of "header2.h". This is a little trickier because you want to delete characters in the middle of the command line, before inserting the correct text. The simplest way to do this is move the cursor to a point just *after* the last character that you want to delete, and then backspace over them. The steps are:
1) Hit the : key to start a new ex command line.
2) Hit the **Up** arrow key or **^O k** repeatedly to move back to the ":e ~/proj1/src/header2.h"command line.
3) Hit the **Left** arrow key five times, or **^O 5 h**, to move the cursor back to the last 'e' character in "header2.h".
4) Hit the **Backspace** key four times to delete the word "head". It will still show on the screen, but elvis will know that it has been deleted. This is the same sort of behavior that elvis (and vi) exhibits when you backspace over newly entered text in input mode.
5) Type **f o o t** to insert "foot" where "head" used to be. At this point, the line should look like ":e ~/proj1/src/footer2.h".
6) Hit **Enter** to submit the revised command line.

4.1.2 The TAB key

The **Tab** key has a special function when you're inputing text into the "Elvis ex history" buffer. It is used for name completion. (Exception: Under MS-DOS, this feature is disabled in order to reduce the size of the program, so it will fit in the lower 640K.)

Name completion works like this: The preceding word is assumed to be a partial name for an ex command, an option, a tag, or a file. The type of name is determined by the context in which it appears -- commands appear at the start of an ex command line, and the others can only occur after certain, specific command names. Elvis searches for all matches of the appropriate type.

If there are multiple matches, then elvis fills in as many characters of the name as possible, and then stops; or, if no additional characters are implied by the matching names, then elvis lists all matching names and redisplays the command line. If there is a single match, then elvis completes the name and appends a tab character or some other appropriate character. If there are no matches, then elvis simply inserts a tab character.

Also, if while entering a :set command you hit the **Tab** key immediately after "*option*=" then elvis will insert the current value of the *option*. You can then edit that value before submitting the command line.

I tried to make elvis smart enough that the **Tab** key will only attempt file/command/option completion in contexts where it makes sense to do so, but that code might not be 100% correct. You can bypass the completion by typing a **Control-V** before the **Tab** key. You can also disable name completion altogether by setting the "Elvis ex history" buffer's inputtab option to "tab", via the following command:

```
:(Elvis ex history)set inputtab=tab
```

or the abbreviated form:

```
:("Eeh)se it=t
```

By default, elvis ignores binary files when performing filename completion. The completebinary option can be used to make elvis include binary files. That's a global option (unlike inputtab which is associated with a specific buffer), so you don't need to specify the buffer name; a simple :set completebinary will set it.

4.2 Syntax and Addressing

In general, ex command lines can begin with an optional window id. This may be followed by an optional buffer id, and then 0, 1, or 2 line addresses, followed by a command name, and perhaps some arguments after that (depending on the command name).

A window ID is typed in as a decimal number followed by a colon character. If you don't supply a window ID (and you almost never will) then it defaults to the window where you typed in the command line. The :buffer command lists the buffers, and shows which one is being edited in which window. Also, the windowid option indicates the ID of the current window.

A buffer ID is given by typing an opening parenthesis, the name of the buffer, and a closing parenthesis. For user buffers, the name of the buffer is usually identical to the name of the file that it corresponds to. For example, a file named ~/.Xdefaults would be loaded into a buffer which could be addressed as (~/.Xdefaults). Elvis also assigns numbers to user buffers, which may be more convenient to type since numbers are generally shorter than names. If ~/.Xdefaults is the first file you've edited since starting elvis, then its buffer could be addressed as (1). The :buffer command shows the number for each user buffer.

Elvis also has several internal buffers, all of which have names that start with "Elvis ", such as (Elvis cut buffer x) and (Elvis error list). The :buffer! command (with a ! suffix) will list them all. For the sake of brevity, elvis allows you to refer to cut buffers as ("x). Similarly, the other internal buffers can be referred to via a " character and the initial letter in each word of the full name, such as ("Eel) for (Elvis error list).

Commands which don't access the text, such as ":quit", don't allow any line addresses. Other commands, such as ":mark", only allow a single line address. Most commands, though, allow two line addresses; the command is applied to all lines between the two specified lines, inclusively. The tables below indicate how many line addresses each command allows.

Line addresses are always optional. The first line address of most commands usually defaults to the current line. The second line address usually defaults to be the same as the first line address. Exceptions are :write, :lpr, :global, and :vglobal, which act on all lines of the file by default, and :!, which acts on no lines by default.

If you use the visual V command to mark a range of lines, and then use the visual : command to execute a single ex command, then the default range affected by the ex command will be the visibly marked text.

Line addresses consist of an absolute part and a relative part. The *absolute part* of a line specifier may be either an explicit line number, a mark, a dot to denote the current line, a dollar sign to denote the last line of the file, or a forward or backward search. An *explicit line number* is simply a decimal number, expressed as a string of digits. A *mark* is typed in as an apostrophe followed by a letter. Marks must be set before they can be used. You can set a mark in visual command mode by typing "m" and a letter, or you can set it in ex command mode via the "mark" command. A *forward search* is typed in as a regular expression surrounded by slash characters; searching begins at the default line. A *backward search* is typed in as a regular expression surrounded by question marks; searching begins at the line before the default line.

If you omit the *absolute part,* then the default line is used.

The *relative part* of a line specifier is typed as a + or - character followed by a decimal number. The number is added to or subtracted from the absolute part of the line specifier to produce the final line number.

As a special case, the % character may be used to specify all lines of the file. It is roughly equivalent to saying 1,$. This can be a handy shortcut.

Here are some addressing examples, using the :p command:

COMMAND	ACTION
:p	print the current line
:37p	print line 37
:'gp	print the line which contains mark g
:/foo/p	print the next line that contains "foo"
:$p	print the last line of the buffer
:20,30p	print lines 20 through 30
:1,$p	print all lines of the buffer

:%p	print all lines of the buffer
:(zot)%p	print all lines of the "zot" buffer
:/foo/-2,+4p	print 5 lines around the next "foo"

The optional addresses are followed by the command name. Command names may be abbreviated. In the sections that follow, the command's full name is given with the optional part enclosed in square brackets.

Some commands allow a '!' character to appear immediately after the command name. The significance of the '!' varies from one command to another, but typically it forces the command to do something dangerous that it would ordinarily refuse to do. For example, :w *file* refuses to overwrite an existing file, but :w! *file* will do it.

Many commands allow (or even require) additional arguments. The descriptions below list which arguments each command accepts with optional commands denoted by square brackets. The most common argument types are:

/regexp/
This is a regular expression. You can use any punctuation character to delimit it, but the '/' character is the most commonly used.

/regexp/newtext/
This is a regular expression followed by replacement text.

count
This is a number - a string of digits. Generally, it is used as the repeat count for certain commands.

cutbuf
This is the name of a cut buffer - a single letter. Elvis also allows (but does not require) a quote character before the letter.

excmds
This is another ex command, or list of ex commands. Traditionally, the whole list of commands had to appear on the same line, delimited by '|' characters. Elvis has the added versatility of allowing a '{' character on the first line, each command on a separate following line, and then '}' on a line by itself to mark the end of the ex command list.

lhs
This is string of characters. If whitespace characters are to be included in it, then they must be quoted by embedding a **^V** character before them.

line
This is a line address, as described earlier.

+line
Some commands which cause a file to be loaded also allow you to specify some other command to be executed after the loading is

complete. To use this feature, you mist give a "+" followed by the command, in between the command name and the file name. Here's an example that loads foo and then moves the cursor to line 40.

```
:e +40 foo
```

Usually the command is just a line number, so this is denoted as "+line" in this documentation. Other commands are allowed though, such as "+/text" to search for text, or "+normal" to force it to use the normal display mode.

Traditionally, commands supplied in this manner weren't allowed to contain whitespace, because that makes parsing the command line harder. This is too limiting, though, so elvis allows you to embed spaces in the command by wrapping the entire deferred command in double-quotes, like this:

```
:e +"set bufdisplay=man" filedb.8
```

mark
This is the name of a mark - a single lowercase letter. Elvis allows (but does not require) an apostrophe before the letter.

rhs
This is a string of characters. If it begins with a whitespace character, then that character must be quoted by embedding a **^V** character in the command line before it. Other whitespace characters in the string do not need to be quoted.

expr
This is an arithmetic expression using the normal syntax.

shellcmd
This is a command line which is passed to the system's command interpreter. Within the command line, the following character substitutions take place, unless preceded by a backslash:

CHARACTER	REPLACED BY
%	Name of current file
#	Name of alternate file
#*n*	Name of file whose bufid=*n*
!	Previous command line
\@	Word at cursor location

Note that the \@ substitution *requires* a backslash. This quirk exists for the sake of backward compatibility - the real vi doesn't perform any substitutions for just plain @.

file or files
This is one or more file name, or a "wildcard" pattern which matches the names of zero or more files. File names are subjected to three levels of processing. First, leading ~

characters and certain other characters are replaced with text, as follows:

SYMBOL	REPLACED BY
~*user*	(Unix only) Replaced by home directory of *user*
~+	Replaced by current working directory
~-	Replaced by previous directory (previousdir)
~	Replaced by home directory (home)
%	Replaced by the name of the current file
#	Replaced by the name of the alternate file
#*n*	Replaced by the filename of buffer with bufid=*n*
(space)	Delimits one file name from another
\`program\`	Run program, interpret its output as filenames

The second stage of processing evaluates each name using the simpler expression syntax. This basically means that expressions of the form **$***NAME* will be replaced with the value of the environment variable named *NAME*. Also, you can use parentheses around option names or more complex expressions. For example, if the user option f contains the name of a file, then you could say ":e (f)" to edit that file.

In either of the first two stages, backslashes may be used to prevent the special symbols from having their usual meaning; they'll be treated as normal text instead. In particular, a backslash-space sequence can be used to give a filename which includes spaces; e.g., to edit "C:\Program Files\foo" you would type ":e C:\Program\ Files\foo". Note that backslashes which are followed by a normal character are simply retained as normal characters, so you rarely need to type a double-backslash when your file name needs only a single backslash.

The third stage of processing checks for "wildcard" characters in the name, and if there are any then the whole name is replaced by the name of each matching file. The exact list of supported wildcards will vary from one operating system to another, but the following are typical:

SYMBOL	MATCHES
*	Any string of characters, of any length
?	Any single character
[a-z]	(Unix only) Any single character from A to Z

In most operating systems, wildcards are only recognized when they occur in the last file name part of a longer pathname. In other words, you can use wildcards for file names, but not in directory names leading up to file names.

Traditionally, vi has used the Unix shell to expand wildcards. However, this interferes with the use of spaces in file names, isn’t easily portable to non-Unix operating systems, and is a potential security hole. So elvis performs all wildcard expansion itself. The only disadvantage of this is that you loose other shell notations such as ‘command‘ and {alt1,alt2}.

Most commands can be followed by a ’|’ character and another ex command. Others can’t. In particular, any command which takes a **excmd** or **shellcmd** argument doesn’t treat ’|’ as a command delimiter.

If a command does treat ’|’ as a delimiter, and you want ’|’ to be treated as part of a command argument, then you’ll need to quote the ’|’ character by preceding it with a backslash or ^V, depending on the command. (Sadly, different commands require different quote characters.)

4.3 Ex Commands, Grouped by Function

4.3.1 The help command itself

ADDRESS	COMMAND	ARGUMENTS
	h[elp]	topic

The *:help* command loads and displays a help file for a given topic. There are several help files, covering a wide variety of topics.

Elvis looks at the topic you supply, and tries to determine whether it is an ex command name, vi keystroke, option name, or something else. Based on this, it generates a hypertext link to the topic in the appropriate help file, and shows the topic in a separate window. Elvis uses the following rules to convert your requested topic into a hypertext reference:

COMMAND	ELVIS’ INTERPRETATION

:help	With no topic, elvis loads the table of contents. This has hypertext links that can lead you to any other topic.
:help ex	Elvis loads the chapter describing ex commands.
:help vi	Elvis loads the chapter describing vi commands.
:help set XXX	If XXX is an option name, elvis will show the description of that option; else it will list groups of all options.
:help :XXX	If XXX is an ex command name, elvis will show its description; else elvis will list groups of all ex commands.
:help XXX	If XXX appears to be a keystroke then elvis will assume it is meant to be a vi command and will show the command's description. Else if it is an option name elvis will show that. Else if it is an ex command, elvis will show that. Else elvis will show this description of the :help command itself.

Although this chart only mentions sections on ex commands, vi commands, and options, there are many others which are only accessible via the table of contents shown by ":help" with no arguments.

All of these help files are HTML documents. Elvis' standard HTML editing facilities are available while you're viewing the help text. Some of the highlights of this are:

* To close this help window, type **ZQ**. Actually, this works for all windows. (You must hold the **Shift** key as you type **ZQ**, because lowercase **zq** does something else entirely: nothing!)
* Any underlined text is a hypertext reference. This means that you can move the cursor onto it, and hit the **Enter** key, and the cursor will move to a topic describing the underlined text.
* To return to your original position after following a hypertext reference, hit **^T** (Control-T).
* The **Tab** key moves the cursor forward to the next hypertext reference.

You can use elvis to print the document via the :lpr command. This assumes you have set the printing options correctly.

NOTE: In addition to the :help command, most versions of elvis also support two aliases which you may find handy. The ":howto *words...*" alias searches for given words in the title lines of a short "howto.html" document. The ":kwic *word*" alias finds every instance of a given word in any section of elvis' documentation, and builds a table showing each instance along with some of the surrounding text; you can then follow hypertext links to the actual location in the manual.

4.3.2 Editing commands

ADDRESS	COMMAND	ARGUMENTS
line	a[ppend][!]	[text]
line	i[nsert][!]	[text]
range	c[hange][!]	[count] [text]
range	d[elete]	[cutbuf] [count]
range	y[ank]	[cutbuf] [count]
line	pu[t]	[cutbuf]
range	co[py]	line
range	m[ove]	line
range	t[o]	line
range	!	shellcmd
range	>	
range	<	
range	j[oin][!]	
	u[ndo]	[count]
	red[o]	[count]

The *:append* command inserts text after the current line. If no new text is supplied on the command line, then elvis will wait for you to type in text; you can then mark the end of the new text by typing a "." (period) on a line by itself. In the real vi, adding a '!' suffix temporarily toggles the autoindent option, but elvis just ignores the '!'.

The *:insert* command inserts text before the current line. Other than that, it is identical to the :append command. In the real vi, adding a '!' suffix temporarily toggles the autoindent option, but elvis just ignores the '!'.

The *:change* command deletes old text lines (copying them into the anonymous cut buffer) and then waits for you to enter new text to replace it. You can then mark the end of the new text by typing a "." (period) on a line by itself. In the real vi, adding a '!' suffix temporarily toggles the autoindent option, but elvis just ignores the '!'.

The *:delete* command copies text into a cut buffer, and then deletes it from the edit buffer. The *:yank* command copies text into a cut buffer but leaves the edit buffer unchanged.

The *:put* command "pastes" text from a cut buffer back into the edit buffer. The cut buffer's contents are inserted after the addressed line. If you want to insert before the first line, you can use address 0 like this:

```
:0put
```

The *:copy* and *:to* commands are identical. They both make a copy of a portion of an edit buffer, and insert that copy at a specific point. The destination line can be specified with an optional buffer name and the full address syntax as described in section 4.2. Consequently, you can use this command to copy part of one edit buffer into another edit buffer. The following example copies an

11-line window from the current buffer onto the end of a buffer named "otherbuf"

:-5,+5t(otherbuf)$

The *:move* command resembles :copy except that *:move* deletes the original text.

The *:!* command allows you to send parts of your edit buffer though some external "filter" program. The output of the program then replaces the original text. For example, this following will sort lines 1 through 10 using the "sort" program.

:1,10!sort

If you use the *:!* command without any line addresses, then elvis will simply execute the program and display its output. This is only guaranteed to work correctly for non-interactive programs; to execute an interactive program you should use the :shell command.

The *:<* and *:>* commands adjust the indentation on the addressed lines. The *:<* command decreases the leading whitespace by the number of spaces indicated in the shiftwidth option, and *:>* does the reverse. You can use multiple < or > characters in a single command to increase the shift amount; for example, *:>>>* shifts text by triple the shiftwidth amount. Normally elvis' versions of these commands will leave blank lines unchanged, but if you append a '!' (as in *:>!*) then the command will affect blank lines in addition to other lines.

The *:join* command joins multiple lines together so they form one long line. Normally it will intelligently decide how much whitespace it should place between lines, depending on the sentenceend, sentencegap, and sentencequote options. When invoked with a '!' suffix (as in :join!), it joins the lines without doing fancy things to whitespace.

The *:undo* command undoes recent changes. The number of undoable changes is controllable on a buffer-by-buffer basis, via the undolevels option. The *:redo* command undoes an undo.

4.3.3 Global edit commands

ADDRESS	COMMAND	ARGUMENTS
range	g[lobal][!]	/regexp/ excmds
range	v[global][!]	/regexp/ excmds
range	s[ubstitute]	/regexp/text/[g|.*n*][x][c][p|l|#]
range	&	[g|.*n*][p|l|#][x][c]
range	~	[g|.*n*][p|l|#][x][c]

The *:global* command searches for lines which contain the /regexp/ and executes the given *excmds* for each matching line. The *:vglobal* command executes the *excmds* for each line which *does not* match the

/regexp/.

In script files, you can supply multiple command lines to a single :global or :vglobal by placing a '{' character on the :global/:vglobal line, following that with any number of command lines, and then finally a '}' character on a line by itself to mark the end. This notation doesn't allow nesting; you can't use {...} inside a larger {...} command list. (Hopefully this limitation will be lifted soon.)

The *:substitute* command searches for the */regexp/* in each line, and replaces the matching text with *newtext.* The interpretation of *newtext* is described in section 5.2

The *newtext* can be followed by a **g** flag to replace all instances in each line. Without the **g** flag, only the first match within each line is changed (unless the gdefault option is set). To replace some other instance in each line, give a decimal point followed by the instance number, such as **.3** to replace the third instance of matching text in each line.

You can also supply a **p** flag. This causes each affected line to be printed (like :p), after all substitutions have been made to that line. Similarly, **l** lists it (like :l), and **#** prints it with a line number (like :nu or :#).

You can also make elvis ask for confirmation before each substitution by appending a **c** flag. The :s command will locate the first match and then exit immediately, but it will leave the window in an unusual input state in which **y** performs a substitution and then moves on to the next match, **n** does not perform the substitution but still moves to the next match, and **Esc** cancels the operation. Most other keys act like **y** in this mode.

NOTE: Elvis doesn't allow the **c** flag to be combined with the :g command. Instead of using ":g/regexp/s//newtext/gc", I suggest you get in the habit of using ":%s/regexp/newtext/gc". However, there is no way to do the more complex ":g/regexp1/s/regexp2/newtext/gc" in elvis at this time.

Elvis supports a special **x** flag. Instead of performing each substitution, elvis will execute the final replacement text as an ex command line. This is used in the implementation of modelines, like this:

```
1,5 s/ex:\(.*\):/\1/x
$-4,$ s/ex:\(.*\):/\1/x
```

The *:&* and *:~* commands both repeat the previous *:substitute* command, discarding any previous flags. The difference between them is that :& uses the regular expression from the previous :s command, but :~ uses the most recent regular expression from any context.

4.3.4 Displaying text

ADDRESS	COMMAND	ARGUMENTS
range	p[rint]	[count]
range	l[ist]	[count]
range	nu[mber]	[count]
range	#	[count]
line	z	[spec]
range	=	

The *:print* command displays lines from the edit buffer. It displays them the normal way -- with tabs expanded and so on.

The *:list* command also displays lines, but it tries to make all non-printing characters visible, and it marks the end of each line with a '$' character.

The *:number* and *:#* commands are identical to each other. They both display lines the normal way except that each line is preceded by its line number.

The *:z* command shows a "window" of lines surrounding the current line. The default size of the "window" is taken from the window option. If a line address is supplied, then it becomes the current line before this command is executed. The *spec* can be one of the following characters; the default is **z+**.

SPEC	OUTPUT STYLE
-	Place the current line at the bottom of the window.
+	Place the current line at the top of the window. Upon completion of this command, the last line output will become the current line.
^	Jump back 2 windows' worth of lines, and then do the equivalent of z+. Note that z+ is like paging forward and z^ is like paging backward.
.	Place the current line in the middle of the window. Upon completion of this command, the last line output will become the current line.
=	Place the current line in the middle of the window, and surround it with lines containing hyphens.

The *:=* command displays the line number of the current line, or the addressed line if given one address. If given a range of addresses, it tells you the line numbers of the two endpoints and the total number of lines in the range.

4.3.5 Tags

ADDRESS	COMMAND	ARGUMENTS
	ta[g][!] stac[k] po[p][!] br[owse][!]	[tag] restrictions

Tags provide a way to associate names with certain places within certain files. Typically, you will run the **ctags** program to create a file named "tags" which describes the location of each function and macro used in the source code for your project. The tag names are the same as the function names, in this case.

In HTML mode, elvis uses the tags commands to follow hypertext links, but we'll generally ignore that in the following discussions.

The *:tag* command performs tag lookup. It reads the "tags" file to locate the named tag. It then loads the source file where that tag is defined, and moves the cursor to the specific point within that buffer where the tag is defined. Elvis' implementation of :tag also allows you to give extra restrictions and hints. There is also a :stag command which creates a new window and moves its cursor to the tag's definition point.

The *:browse* command extracts selected tags from the tags file, constructs an HTML document listing those tags (with hypertext links to their definition points inside your source code) and displays it in the current window. There is also a :sbrowse command which displays the same list in a new window. If invoked with no args, they browse all tags in the current file. If invoked with a '!' suffix, they browse *all* tags. See chapter 14. Tags for a full description of restrictions and hints, and browsing.

Before moving the cursor, elvis will save the old cursor position on a stack. You can use the *:stack* command to display the contents of that stack. Each window has an independent stack.

The *:pop* command pops a cursor position off the stack, restoring the cursor to its previous position. When you're browsing though source code, you will typically use *:tag* to go deeper into the call tree, and *:pop* to come back out again.

In HTML mode, these all work the same except that *:tag* expects to be given an URL instead of a tag name. URLs don't depend on having a "tags" file, so the "tags" file is ignored when in HTML mode. Elvis doesn't support any network protocols, so its URLs can only consist of a file name and/or a #label. The following example would move the cursor to the start of this section:

```
:tag elvisopt.html#TAGS
```

4.3.6 File I/O commands

ADDRESS	COMMAND	ARGUMENTS

line	r[ead]	file | !shellcmd
range	w[rite][!]	[file | >>file | !shellcmd]
range	lp[r][!]	[file | >>file | !shellcmd]

The *:read* command reads a file or external program, and inserts the new text into the edit buffer after the addressed line. If you don't explicitly give a line address, then the text will be inserted after the current line. To insert the file's contents into the top of the buffer (before line 1), you should specify line 0. For example, to insert the contents of "foo.txt" before line 1, you would give the command...
:0 read foo.txt

The *:write* command writes either the entire edit buffer (if no address range is given) or a part of it (if a range is given) out to either a file or an external filter program. If you don't specify the output file or external command, then elvis will assume it should write to the file that the buffer was originally loaded from.

Elvis will normally prevent you from overwriting existing files. (The exact details of this protection depend on the edited, filename, newfile, readonly, and writeany options.) If you want to force elvis to overwrite an existing file, you can append a "!" to the end of the command name, but before the file name. In order to avoid ambiguity, *there must not be any whitespace between the "write" command name and the "!" character* when you want to overwrite an existing file. Conversely, when writing to an external program there *should* be whitespace before the "!" that marks the start of the program's command line. The ">>file" notation tells elvis to append to "file" instead of overwriting it.

The *:lpr* command sends text to the printer. It is similar to :write except that :lpr formats the buffer contents as defined by the bufdisplay option and the printing options. If no output file or external program is specified, then the printer output is sent to the file or external program specified by the lpout option.

4.3.7 The args list, and selecting a file to edit

ADDRESS	COMMAND	ARGUMENTS
	ar[gs]	[file...]
	n[ext][!]	[file...]
	N[ext][!]	
	pre[vious][!]	
	rew[ind][!]	
	la[st]	
	wn[ext][!]	
	f[ile]	[file]
	e[dit][!]	[+line] [file]
	ex[!]	[+line] [file]
	vi[sual][!]	[+line] [file]
	o[pen][!]	[+line] [file]

The "args list" is a list of file names. It provides an easy way to edit a whole series of files, one at a time. Initially, it contains any file names that you named on the command line when you invoked elvis.

The *:args* command displays the args list, with the current file name enclosed in brackets. You can also use :args to replace the args list with a new set of files; this has no effect on whatever file you're editing at that time, but it will affect any :next commands that you give later.

The *:next* command switches to the next file in the args list. This means it loads the next file from the args list into an edit buffer, and makes that edit buffer be the current buffer for this window. You can also give a new args list on the :next command line; this acts like a :args command to set the args list, followed by an argumentless :next command to load the next (first) file in that list.

The *:Next* (with a capital "N") and *:previous* commands are identical to each other. They both move backwards through the args list.

The *:rewind* and *:last* commands switch to the first and last files in the args list, respectively.

The *:wnext* command is like a :write command followed by a :next command. It saves any changes made to the current file before switching to the next file. (The autowrite option offers a better alternative.)

The *:file* command displays information about the current buffer. It can also be used to change the filename associated with this buffer.

The *:edit* and *:ex* commands are identical to each other. They both switch to a new file, or if no file is named then they reread the current file. This has no effect on the args list.

The *:visual* and *:open* commands switch to a new file if one is named; otherwise they continue to use the current buffer *without* reloading it from the original file. These commands have the side-effect of switching the window mode from ex mode to either the normal visual mode or the uglier "open" mode, respectively. "Open" mode allows you to use all of the visual commands, but it only displays a single line (the line that the cursor is on) at the bottom of the screen. The sole advantage that "open" mode has over "visual" mode is that "open" mode doesn't need to know what kind of terminal you're using.

4.3.8 Quitting

ADDRESS	COMMAND	ARGUMENTS
	cl[ose][!] q[uit][!]	

	wq[uit][!] x[it][!] qa[ll][!] pres[erve]	[file] [file]

Except for :qall, all of these commands attempt to close the current window without losing any changes. When the last window is closed, elvis exits. The differences between these commands concern how modified buffers are handled. In the discussions below, it is assumed that tempsession is True and the buffer's retain option is False, which is usually the case.

The *:close* command is the simplest. If the current window is the only window and one or more buffers have been modified but not yet saved, then :close will fail; otherwise the current window will be closed. The visual ^Wq command uses this command internally. If the window's buffer was modified, then elvis will just have a modified buffer lying around, which may or may not be visible in some other window. That's okay. The other quitting commands won't allow you to lose that buffer accidentally. You can make some other window view that buffer by giving that buffer's name in parentheses on an ex command line in that other window.

The *:quit* command fails if the current buffer has been modified. If you wish to abandon the changes made to the current buffer, you can add a "!" to the end of the command name; this has the effect of turning off the buffer's modified flag.

The *:xit* command saves the file if it has been modified, and then closes the window. The visual ZZ command uses this command internally.

The *:wquit* command saves the file regardless of whether it has been modified, and then closes the window.

The *:qall* command tries to close all of the windows at once. It is equivalent to giving the :quit command in each window.

The *:preserve* command closes all windows and exits, but it doesn't delete the session file. You can restart the same edit session later by giving the command...

elvis -s*sessionfile*

...where *sessionfile* is the name of the session file, usually "/var/tmp/elvis1.ses". You may want to check the value of the session option first, just to make sure.

4.3.9 Scripts and macros

ADDRESS	COMMAND	ARGUMENTS
	@ so[urce][!]	cutbuf file

ADDRESS	COMMAND	ARGUMENTS
	saf[er][!]	file
	al[ias]	[name [excmds]]
	unal[ias][!]	name

The *:@* command executes the contents of a cut buffer as a series of ex command lines.

The *:source* command reads a file, and executes its contents as a series of ex commands. Normally, elvis would issue an error message if the requested file didn't exist but when a "!" is appended to the command name, elvis will silently ignore the command if it doesn't exist.

The *:safer* command is exactly like :source, except that :safer will temporarily set the safer option while it is executing the commands. You should use :safer instead of :source when it is possible that the file to be executed could contain potentially harmful commands. For example, the default "elvis.ini" file uses :source to execute the ".exrc" file in your home directory since it is presumably secure, but :safer is used to execute the ".exrc" file in the current directory since it could have been created by anybody. As with *:source!*, invoking *:safer!* (with a '!' suffix) prevents elvis from complaining about nonexistent script files.

The *:alias* and *:unalias* commands manipulate the alias list. (See the Tips section of the manual for a discussion of aliases.) With no arguments, :alias displays all aliases. When given a name but no commands, :alias displays the complete definition of the named alias. When given a name and commands, :alias defines (or redefines) an alias. The :unalias command deletes the alias with a given name.

4.3.10 Working with a compiler

ADDRESS	COMMAND	ARGUMENTS
	cc[!]	[args]
	mak[e][!]	[args]
	er[rlist][!]	[file]

If you use elvis to edit source code for programs, then you can have elvis read the output of your compiler and parse that output for error messages. When elvis finds an error message, it can move the cursor to the file and line number where the error was reported.

To parse the compiler output, elvis first breaks the output into lines. Each line is then broken into words. If a word looks like a number, then it is assumed to be a line number. If a word looks like the name of an existing file, then it is assumed to be a file name. Any line which contains both a line number and a file name is treated as an error report (with the remainder of the line serving as a description of the error); lines which don't have both of these are simply ignored.

The *:cc* and *:make* commands use the ccprg and makeprg options, respectively, to run your compiler or "make" utility, and collect the output. Elvis will then move the cursor to where the first error was detected. (If there were no errors, elvis will say so and leave the cursor unchanged.)

After that, the *:errlist* command can be used repeatedly to move to each successive error. You can also use the :errlist command with a file name argument to load a new batch of error messages from a file; the cursor is then moved to the first error in that batch.

4.3.11 Built-in calculator

ADDRESS	COMMAND	ARGUMENTS
	ca[lculate] ev[al]	expr expr

Elvis has a built-in calculator which uses a C-like syntax. It is described in section 12: Arithmetic Expressions. The :if and :let commands also use the calculator.

The *:calculate* command evaluates an expression and displays the result.

The *:eval* command evaluates an expression using the simpler syntax (which basically means that text outside of parentheses is left alone), and then executes the result as an ex command line. This provides a way to use expressions with commands which would not ordinarily use expressions. For example, the following command line inserts the value the elvispath option into the current edit buffer.
:eval insert elvispath=(elvispath)

Note: There is a hardcoded limit of (normally) 1023 characters for the result of any expression. This limit will sometimes impact the use of :eval. For example, if your $EXINIT environment variable is longer than 1023 characters then elvis will be unable to interpret it during initialization.

4.3.12 Buffer commands

ADDRESS	COMMAND	ARGUMENTS
	al[l][!] b[uffer][!] (bb[rowse][!] sbb[rowse][!]	excmds [buffer] buffer

The *:all* command applies a given ex command line to each edit buffer in turn. Normally the command is applied just to the user edit buffers, but if you append a "!" to the command name, then the ex command line is applied to internal buffers as well. For example, the following sets the "bufdisplay" option of all user edit buffers: :all set bufdisplay=normal

In script files, you can supply multiple command lines to a single :all commands by placing a '{' character on the :all line, following that with any number of command lines, and then finally a '}' character on a line by itself to mark the end. This notation doesn't allow nesting; you can't use {...} inside a larger {...} command list. (Hopefully this limitation will be lifted soon.)

The *:buffer* command lists either all user edit buffers, or (when "!" is appended to the command name) *all* buffers including internal ones. If the buffer is being edited in one or more windows, then the window ID is also displayed. Buffers which have been modified will be marked with an asterisk.

You can also use the :buffer command to make the current window display a different buffer.

The *:(buffer* notation causes the current window to display the named buffer, instead of the current buffer. This isn't really a command; it is part of an address. Whenever you give an address without specifying a command, elvis moves the cursor to the addressed line. In this particular case, we're addressing the most recently changed line of a given buffer, so that's where the cursor is moved to. For more information, see the discussion of Buffer IDs earlier in this chapter (in the discussion of addresses).

The *:bbrowse* and *:sbbrowse* commands create an HTML document which lists the names of all user buffers (or, when a '!' is appended to the command name, *all* buffers including internal buffers). You can then go to one of the buffers just by following the hypertext link. The difference between these two commands is that :bbrowse displays the list in the current window, but :sbbrowse creates a new window to display it.

4.3.13 Window commands

ADDRESS	COMMAND	ARGUMENTS
	sp[lit]	[+line] [file | !shellcmd]
	new	
	sne[w]	
	sn[ext]	[file...]
	sN[ext]	
	sre[wind]	
	sl[ast]	
	sta[g]	[tag]
	sb[rowse]	restrictions
	sa[ll]	
	wi[ndow]	[+[+] | -[-] | number | buffer]
	di[splay]	[modename [language]]
	no[rmal]	

The *:split* command creates a new window. If you supply a file name, then it will load that file into an edit buffer and the new window will show that buffer. If you supply a shell command line preceded by a '!' character, then it will create an untitled buffer, and read the output of that command line into the buffer. Otherwise, the new window will show the same buffer as the current window.

The *:new* and *:snew* commands are identical to each other. They both create a new empty buffer, and then create a new window to show that buffer.

The *:snext, :sNext, :srewind, :slast, :stag,* and *:sbrowse* commands resemble the :next, :Next, :rewind, :last, :tag, and :browse commands, respectively, except that these "s" versions create a new window for the newly loaded file, and leave the current window unchanged.

The *:sall* command creates a new window for any files named in the args list, which don't already have a window. (See section 4.3.7: The args list... for a discussion of the args list.)

The *:window* command either lists all windows (when invoked with no arguments) or switches to a given window. You can specify which to switch to by giving one of the following arguments.

ARGUMENT	MEANING
+	Switch to the next window, like ^Wk
++	Switch to the next window, wrapping like ^W^W
-	Switch to the previous window, like ^Wj
--	Switch to the previous window, wrapping
number	Switch to the window whose windowid=number
buffer	Switch to the window editing the named buffer

The *:display* command switches the window to a new display mode, overriding the value of the bufdisplay option. The display option indicates the current display mode. If you omit the new modename, then the :display command will list all supported display modes, with an asterisk next to the current mode. The "syntax" mode allows you to specify which language's syntax it is supposed to use; if you don't specify a language, elvis will guess the language from the file name's extension.

The *:normal* command is equivalent to ":display normal". It can be abbreviated to ":no", which is certainly easier to type than ":dis normal".

4.3.14 Configuration

ADDRESS	COMMAND	ARGUMENTS
	ab[breviate][!]	[lhs rhs]

<table>
<tr><td></td><td>una[bbreviate][!]</td><td>lhs</td></tr>
<tr><td></td><td>map[!]</td><td>[lhs rhs]</td></tr>
<tr><td></td><td>unm[ap][!]</td><td>lhs</td></tr>
<tr><td></td><td>bre[ak][!]</td><td>lhs</td></tr>
<tr><td></td><td>unb[reak][!]</td><td>lhs</td></tr>
<tr><td></td><td>dig[raph][!]</td><td>[lhs [rhs]]</td></tr>
<tr><td></td><td>col[or]</td><td>[font color ["on" color]]</td></tr>
<tr><td></td><td>gu[i]</td><td>text</td></tr>
<tr><td></td><td>se[t][!]</td><td>[option=value | option? | all]</td></tr>
<tr><td></td><td>lo[cal][!]</td><td>[option=value | option]</td></tr>
<tr><td></td><td>le[t][!]</td><td>option=expr</td></tr>
<tr><td></td><td>if</td><td>expr</td></tr>
<tr><td></td><td>th[en]</td><td>excmds</td></tr>
<tr><td></td><td>el[se]</td><td>excmds</td></tr>
<tr><td></td><td>try</td><td>excmds</td></tr>
<tr><td></td><td>wh[ile]</td><td>expr</td></tr>
<tr><td></td><td>do</td><td>excmds</td></tr>
<tr><td></td><td>switch</td><td>expr</td></tr>
<tr><td></td><td>case</td><td>value [excmds]</td></tr>
<tr><td></td><td>default</td><td>excmds</td></tr>
<tr><td></td><td>mk[exrc][!]</td><td>[file]</td></tr>
</table>

The *:abbreviate* and *:unabbreviate* commands add and remove entries to the abbreviation table, respectively. Also, the :abbreviate command can be used with no arguments to list the current contents of the abbreviation table. For a discussion of abbreviations, see section 3.3: Abbreviations. Normal abbreviations are only active while you're typing in a normal text buffer; adding a '!' suffix to the command name causes the macro to be active while you're entering ex command lines.

The *:map* and *:unmap* commands add and remove entries to the map tables, respectively. When the :map command is given without any arguments, it lists the contents of a map table.

There are two map tables. When a "!" is appended to the command name, these commands use the table that applies to input mode; without the "!" these commands use the table that applied to visual command mode.

The primary purpose of map tables is to assign actions to the cursor keypad and the function keys. Each of these keys sends an arbitrary but distinctive sequence of characters when pressed. The map tables are used to convert these arbitrary character sequences into command keystrokes that elvis can do something useful with. For example, arrow keys are normally mapped to the h, j, k, and l commands.

The first argument to :map is the raw character sequence sent by a key, and the remaining arguments are the characters that elvis should pretend you pressed. This can be either a literal sequence of characters, or a gui-dependent symbol representing a particular keystroke. See the User Interfaces chapter for lists of keystrokes. Also, function keys can usually be denoted by #1 for the <F1> key, #2 for the <F2> key, and so on.

The second argument is character sequence that elvis should pretend you typed whenever the raw characters are received. This may be preceded by the word "visual" which causes the remaining argument characters to be processed as visual commands, even if the key is pressed in input mode. This trick is used to allow the cursor to be moved via the arrow keys when in input mode.

The *:break* and *:unbreak* commands set and reset the breakpoint flag for a given macro, respectively. Using a '!' suffix causes the breakpoint to be set for an input-mode map. This is used for debugging macros, as described in section 16.3: How to debug macros. If a macro has its breakpoint flag set, and the maptrace option is set to **run**, then when that map is encountered elvis will automatically switch maptrace to **step** mode.

The *:digraph* command manipulates the digraph table. (See section 3.2: Digraphs for a discussion on digraphs.) With no arguments, it lists the digraph table. With one argument, it removes the given digraph from the table. With two arguments, it adds the given digraph to the table, or if the same two ASCII characters are already in the table then it alters the existing entry.

Normally, the :digraph command sets the most significant bit in the last argument's character. That way you don't need to be able to type a non-ASCII character on your keyboard in order to enter it into the table; you can type the ASCII equivalent and allow elvis to convert it to non-ASCII before storing the digraph. If you don't want elvis to set the most significant bit, then append a "!" to the end of the command name.

The *:color* command allows you to choose a color to use for displaying each font. Some user interfaces don't support this. The ones that do will vary in the color names that they support. The termcap interface supports **black**, **white**, **gray**, **red**, **green**, **blue**, **brown**, **yellow**, **magenta**, and **cyan**, plus **light** or **bright** versions of most of those. The windows interface supports the same colors, except that it is pickier: it doesn't allow spaces, and only "light" is accepted, such as "lightblue". The x11 interface supports all standard X color names.

The first argument should be the name of the font to change. This can be "normal", "bold", "emphasized", "italic", "underlined", or "fixed". Some user interfaces may also support "standout", "cursor", "scrollbar", and/or "toolbar". All of these can be either spelled out completely, or abbreviated to the first letter. (Currently no user interface supports both "standout" and "scrollbar" so there is no ambiguity.) If you omit the font name, then "normal" is assumed. The termcap interface requires you to assign a "normal" color before any of the other fonts.

You can specify an optional background color. The word "on" is used to delimit the foreground color name from the background color name. For example, the command ":color yellow on blue" causes normal text to be displayed as yellow characters on a blue background.

The x11 user interface allows you to specify both the foreground and background color for the cursor. The cursor is drawn in the foreground color normally, but the background color if elvis owns the current X selection.

The *:gui* command provides a way to pass unusual commands to the user interface. Currently, the only user interface which uses this is the "x11" interface, which uses it to configure the toolbar.

The *:set* command allows you to examine or change the values of options. Using *:set!* (with a "!" at the end of the command name) causes it to include the group name of any option that is output. In addition, "!" inhibits the setting of any option's "modified" flag, which will then prevent it from being output by a later argumentless *:set* command.

With no arguments, :set lists the names and values of any options that have been altered or are of frequent interest. If given the argument "all" it will list the names and values of most (but not really all) options. If given the name of an option followed by a "?" character, :set will output the option's name and value. If given the name of a group of options, followed by a "?" character, :set will output the names and values of all options in that group.

To turn a Boolean option on, just give the name of the option. You can turn it off by adding the prefix "no" to the option name, and you can negate it by adding the "neg" prefix to its name.

To change the value of a non-Boolean option, give the name followed *immediately* by an "=" and the new value. If the new value contains whitespace, you should either enclose the entire value in quotes, or precede each whitespace character with a backslash.

If you give the name of a non-Boolean option, without either "=*value*" or "?", then elvis will display its value.

EXAMPLE	WHAT IT DOES
:set	display names & values of changed/interesting options
:set all	display names & values of most POSIX-compliant options
:set ts?	display name & value of the tabstop option
:set lp?	display names & values of all printing options
:set ts=4	set value of the tabstop option to 4
:set ai	turn on the autoindent option
:set noai	turn off the autoindent option
:set negai	toggle the autoindent option

The *:local* command is similar to :set, and is intended to be used in aliases and scripts. In addition to setting options' values, it also pushes the old values onto a stack; the old values are automatically restored at the end of the alias or script. Another difference is that where :set would output an option, :local merely pushes its old value, without outputting or changing the option's value. This means that you can save a non-Boolean option simply by mentioning its name on a :local command line; Boolean options can also be saved without altering them, but you must put a question mark after the option's name.

Here's a simple alias which uses :local. It totals the numbers in all lines from the current line forward to the next line which contains the "total:", and stores the total in the "total:" line.

```
:alias total {
        local nowrapscan ignorecase t=0
        .,/total:/-1 s/\d\+/let t=t+&/x
        eval /tota:l/ s/total:.*/total: (t)/
}
```

The *:let* command computes a new value for an option. The :let command should be followed by the name of an option, then an "=" sign, and then an expression that produces the new value. Note that even Boolean options use the "=" notation here. When invoked as *:let!* (with a '!' suffix), elvis won't set the option's "changed" flag so it won't be output by an argumentless :set command.

The *:if* command evaluates an expression, and sets an internal variable according to whether the result was true or false. Later, the *:then* and *:else* commands can be used to test that variable, and conditionally execute other ex commands.

Note that after an :if command, any other ex commands which don't start with :then or :else will be executed unconditionally.

In aliases or script files, you can supply multiple command lines to a single :then or :else by placing a '{' character on the :then/:else line, following that with any number of command lines, and then finally a '}' character on a line by itself to mark the end. The following example demonstrates this syntax, and also shows that you can safely use :if inside a :then or :else command:

```
:if i <= 0
:then {
        if i == 0
        then echo zero
        else echo negative
}
:else echo positive
```

The *:try* command executes its argument text as an ex command line. Regardless of whether that command line succeeds or fails, the :try command itself always succeeds. That's significant because a command fails, all pending aliases, macros, and scripts are cancelled; :try prevents that. Error messages and warning messages are disabled while the command line runs. Afterward, the then/else is set to indicate whether the command line succeeded. This command is useful for implementing specialized error handing in an alias or script. The following example will search for "foo"; if there is no "foo" then it will search for "bar":

```
:try /foo
:else /bar
```

The *:while* command stores an expression. It should be followed by a *:do* command; the :do command repeatedly evaluates the expression, and as long as the result is true it executes the commands which follow the :do. The following example counts from 1 to 10:

```
:let i=1
:while i <= 10
:do {
        calc i
        let i=i+1
}
```

The *:switch* command evaluates an expression and stores the result. The *:case* command compares that result to a given value, and executes an ex command line if it matches. If you omit the command line then it will carry forward to the next :case which does have an ex command line; this allows multiple cases to share a single ex command line. The *:default* command executes its ex command line if no previous case was matched. Here's an example:

```
:switch os
:case unix
:case win32 echo Elvis has graphical and text-mode interfaces.
:case os2 {
 echo Elvis has been ported natively as a text-mode program.
 echo You can also compile it to use the X11 interface from
 echo Unix, but you need the EMX libraries to run it.
}
:case msdos echo Elvis is slow and ugly under DOS.
:default echo How are you even running this?
```

Notice that there is no punctuation after the case value, and that there is no "break" or "endswitch" command. This example says "Elvis has graphical and text mode interfaces" for both the "unix" and "win32" cases; the "unix" case has no command, so it falls through to the "win32" case.

The :if/:then/:else, :while/:do, and :switch/:case/:default command structures all permit nesting. I.e., the commands in a :then command can't affect the "if" variable to cause the :else command to also be executed.

The *:mkexrc* command creates a file containing ex commands which recreate the current map, abbreviation, and digraph tables, and also sets any options which have been changed. Basically it stores your current configuration in a file which you can later :source to restore your configuration. If you don't specify a filename, then it will write to ".exrc" or "elvis.rc" in the current directory.

NOTE: The *:mkexrc* command does not store information for :alias or :gui commands. This is expected to be added in a later version of elvis.

4.3.15 Miscellaneous

ADDRESS	COMMAND	ARGUMENTS
	"	text
	cd[!]	[directory]
	chd[ir][!]	[directory]
	ec[ho]	text
	me[ssage]	text
	wa[rning]	text
	erro[r]	text
	sh[ell]	
	st[op][!]	
	sus[pend][!]	
	ve[rsion]	
line	go[to]	
line	ma[rk]	mark
line	k	mark
	{	

The " command causes the remainder of the line to be ignored. It is used for inserting comments into ex scripts.

The *:cd* and *:chdir* commands are identical. They both change the current working directory for elvis and all its windows. If you don't specify a new directory name then elvis will switch to your home directory.

The *:echo* command displays its arguments as a message. This may be useful in ex scripts.

The *:message*, *:warning*, and *:error* commands all output their arguments as a message. The command name indicates the importance of the message. This is diffs from :echo as follows: the messages are translated via the elvis.msg file, then evaluated using the simpler syntax, and finally stuffed into the message queue. The message queue collects all messages, and outputs them immediately before waiting for the next keystroke. Also, the :error command has the side-effect of terminating any macros, aliases, or scripts.

The *:shell* command starts up an interactive shell (command-line interpreter). Elvis will be suspended while the shell executes. (Exception: the "x11" GUI runs the shell in a separate xterm window. The elvis and the shell can then run simultaneously.)

The *:stop* and *:suspend* commands are identical to each other. If the operating system and user interface support it, they will suspend elvis and resume the shell that started elvis. (This is like hitting **^Z** on many UNIX systems.) If the OS or GUI don't support it, then elvis will generally treat these commands as synonyms for the :shell command.

The *:version* command identifies this version number of elvis, and displays credits.

The *:goto* moves the cursor to the addressed line. This is the only command which can be abbreviated down to zero characters, so if you type in a line containing just a line address, then elvis will treat that as a :goto command.

The *:mark* and *:k* commands are identical to each other. They set a named mark to equal the addressed line, or the current line if no address was given.

The *{ commands }* notation isn't really a command; it is a feature of elvis' syntax which allows you to pass several command lines to a command which normally expects a single command line as its argument. It is supported by the :global, :vglobal, :all, :then, and :else commands. Instead of placing the argument command at the end of one of those command lines, you can place a single '{' character there. That should be followed by one or more command lines, and terminated by a '}' on a line by itself.

4.4 Alphabetical list of ex commands

ADDRESS	COMMAND	ARGUMENTS
	ab[breviate][!]	[lhs rhs]
	al[l][!]	excmds
line	a[ppend][!]	[text]
	ar[gs]	[file...]
	bb[rowse]	
	bre[ak][!]	lhs
	br[owse][!]	restrictions
	b[uffer][!]	[buffer]
	ca[lculate]	expr
	cc[!]	[args]
	cd[!]	[directory]
range	c[hange][!]	[count] [text]
	chd[ir][!]	[directory]
	cl[ose][!]	
	col[or]	[font color ["on" color]]
range	co[py]	line
range	d[elete]	[cutbuf] [count]
	dig[raph][!]	[lhs [rhs]]
	di[splay]	[modename [language]]
	do	excmds
	ec[ho]	text
	e[dit][!]	[+line] [file]
	el[se]	excmds
	er[rlist][!]	[file]
	erro[r]	text
	ev[al]	expr
	ex[!]	[+line] [file]
	f[ile]	[file]
range	g[lobal][!]	/regexp/ excmds
line	go[to]	
	gu[i]	text

<table>
<tr><td></td><td>h[elp]</td><td>topic</td></tr>
<tr><td></td><td>if</td><td>expr</td></tr>
<tr><td>line</td><td>i[nsert][!]</td><td>[text]</td></tr>
<tr><td>range</td><td>j[oin][!]</td><td></td></tr>
<tr><td>line</td><td>k</td><td>mark</td></tr>
<tr><td></td><td>la[st]</td><td></td></tr>
<tr><td></td><td>le[t][!]</td><td>option=expr</td></tr>
<tr><td>range</td><td>l[ist]</td><td>[count]</td></tr>
<tr><td></td><td>lo[cal][!]</td><td>[option=value | option]</td></tr>
<tr><td>range</td><td>lp[r][!]</td><td>[file | >>file | !shellcmd]</td></tr>
<tr><td></td><td>mak[e][!]</td><td>[args]</td></tr>
<tr><td></td><td>map[!]</td><td>[lhs rhs]</td></tr>
<tr><td>line</td><td>ma[rk]</td><td>mark</td></tr>
<tr><td></td><td>me[ssage]</td><td>text</td></tr>
<tr><td></td><td>mk[exrc][!]</td><td>[file]</td></tr>
<tr><td>range</td><td>m[ove]</td><td>line</td></tr>
<tr><td></td><td>new</td><td></td></tr>
<tr><td></td><td>n[ext][!]</td><td>[file...]</td></tr>
<tr><td></td><td>N[ext][!]</td><td></td></tr>
<tr><td></td><td>no[rmal]</td><td></td></tr>
<tr><td>range</td><td>nu[mber]</td><td>[count]</td></tr>
<tr><td></td><td>o[pen][!]</td><td>[+line] [file]</td></tr>
<tr><td></td><td>po[p][!]</td><td></td></tr>
<tr><td></td><td>pre[vious][!]</td><td></td></tr>
<tr><td>range</td><td>p[rint]</td><td>[count]</td></tr>
<tr><td>line</td><td>pu[t]</td><td>[cutbuf]</td></tr>
<tr><td></td><td>qa[ll][!]</td><td></td></tr>
<tr><td></td><td>q[uit][!]</td><td></td></tr>
<tr><td>line</td><td>r[ead]</td><td>file | !shellcmd</td></tr>
<tr><td></td><td>red[o]</td><td>[count]</td></tr>
<tr><td></td><td>rew[ind][!]</td><td></td></tr>
<tr><td></td><td>sN[ext]</td><td></td></tr>
<tr><td></td><td>saf[er][!]</td><td>file</td></tr>
<tr><td></td><td>sa[ll]</td><td></td></tr>
<tr><td></td><td>sbb[rowse]</td><td></td></tr>
<tr><td></td><td>sb[rowse]</td><td>restrictions</td></tr>
<tr><td></td><td>se[t][!]</td><td>[option=value | option? | all]</td></tr>
<tr><td></td><td>sh[ell]</td><td></td></tr>
<tr><td></td><td>sl[ast]</td><td></td></tr>
<tr><td></td><td>sne[w]</td><td></td></tr>
<tr><td></td><td>sn[ext]</td><td>[file...]</td></tr>
<tr><td></td><td>so[urce][!]</td><td>file</td></tr>
<tr><td></td><td>sp[lit]</td><td>[file | !shellcmd]</td></tr>
<tr><td></td><td>sre[wind][!]</td><td></td></tr>
<tr><td></td><td>stac[k]</td><td></td></tr>
<tr><td></td><td>sta[g]</td><td>[tag]</td></tr>
<tr><td></td><td>st[op][!]</td><td></td></tr>
<tr><td>range</td><td>s[ubstitute]</td><td>/regexp/newtext/[g][p][x][count]</td></tr>
<tr><td></td><td>sus[pend][!]</td><td></td></tr>
<tr><td></td><td>ta[g][!]</td><td>[tag]</td></tr>
<tr><td></td><td>th[en]</td><td>excmds</td></tr>
<tr><td>range</td><td>t[o]</td><td>line</td></tr>
<tr><td></td><td>try</td><td>excmds</td></tr>
<tr><td></td><td>una[bbreviate][!]</td><td>lhs</td></tr>
<tr><td></td><td>unb[reak][!]</td><td>lhs</td></tr>
<tr><td></td><td>u[ndo]</td><td>[count]</td></tr>
</table>

<table>
<tr><td></td><td>unm[ap][!]</td><td>lhs</td></tr>
<tr><td></td><td>ve[rsion]</td><td></td></tr>
<tr><td>range</td><td>v[global][!]</td><td>/regexp/ excmds</td></tr>
<tr><td></td><td>vi[sual][!]</td><td>[+line] [file]</td></tr>
<tr><td></td><td>wa[rning]</td><td>text</td></tr>
<tr><td></td><td>wh[ile]</td><td>expr</td></tr>
<tr><td></td><td>wi[ndow]</td><td>[+ | - | number | buffer]</td></tr>
<tr><td></td><td>wn[ext][!]</td><td></td></tr>
<tr><td></td><td>wq[uit][!]</td><td>[file]</td></tr>
<tr><td>range</td><td>w[rite][!]</td><td>[file | >>file | !shellcmd]</td></tr>
<tr><td></td><td>x[it][!]</td><td>[file]</td></tr>
<tr><td>range</td><td>y[ank]</td><td>[cutbuf] [count]</td></tr>
<tr><td>line</td><td>z</td><td>[spec]</td></tr>
<tr><td>range</td><td>!</td><td>shellcmd</td></tr>
<tr><td></td><td>"</td><td>text</td></tr>
<tr><td>range</td><td>#</td><td>[count]</td></tr>
<tr><td>range</td><td>&</td><td></td></tr>
<tr><td></td><td>(</td><td>buffer</td></tr>
<tr><td>range</td><td><</td><td></td></tr>
<tr><td>range</td><td>=</td><td></td></tr>
<tr><td>range</td><td>></td><td></td></tr>
<tr><td></td><td>@</td><td>cutbuf</td></tr>
<tr><td></td><td>{</td><td></td></tr>
<tr><td>range</td><td>~</td><td></td></tr>
</table>