Ex Reference Manual
Version 1.1 − November, 1977

William N. Joy

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

Ex is a UNIX text editor, based on and largely compatible with the standard UNIX editor *ed*. *Ex* is a line oriented editor and has a *command* mode similar to *ed*. *Ex* also has an *open* mode which allows intraline editing on video terminals, and a *visual* mode for screen oriented editing on cursor-addressible terminals such as the LSI ADM-3A and HP 2645. *Ex* gives a great deal of feedback to the user prompting for command input, indicating the scope of changes performed by commands, and giving diagnostics for all error conditions. For more experienced users, *ex* can be made more *terse*. The *ex* user is protected against accidental loss of work by the *undo* command, which can reverse the effect of the last buffer modifying command, and by sensibility restrictions on the *write* command, which prevent loss of the current file and the accidental overwriting of other files. *Ex* has a recovery mechanism which allows work to be saved to within a few lines of changes after system or editor crashes.

The *Reference Manual* provides a concise description of all features of *ex*, summarizing commands, command variants, options and *open* and *visual* modes.

17 January 2004

Ex Reference Manual
Version 1.1 − November, 1977

William N. Joy

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

Introduction

The reference manual summarizes, in a concise form, the features of the text editor *ex.*

History of the editor

Ex is heavily based on the text editor *ed*. The first versions of *ex* were modifications of a text editor *em* developed at Queen Mary's College in England. *Em* was a modified *ed* which had some added features which were useful on high-speed terminals. The earlier versions of *ex* also included features from the modified *ed* in use at UCLA, and the ideas of the present author and Charles Haley, who implemented most of the modifications to *em* which resulted in these early versions of *ex*. Versions of *ex* have been in use since September, 1976. Version 1.1 of *ex* results from a redesign of *ex* implemented by the present author in the summer and fall of 1977.

Acknowledgements

The author would like to thank Chuck Haley, who collaborated on the earlier versions of *ex* and acted as mentor for the design of this version; Bruce Englar, who stimulated the redesign of *ex* and convinced the author of the worth of the intraline editing facilities; and his faculty advisor Susan L. Graham. In addition, a large number of people have contributed ideas to the development of *ex*, aided in its debugging and in the preparation of documentation. The author would like to thank Eric Allman, Ricki Blau, Rich Blomseth, Clint Gilliam, Steve Glanville, Ed Gould, Mike Harrison, James Joyce, Howard Katseff, Ivan Maltz, Doug Merritt, David Mosher, Dick Peters, Bill Rowan, Genji Schmeder, Eric Schmidt, Jeff Schriebman, Kurt Shoens, Bob Tidd, Bob Toxen, Mike Ubell, and Vance Vaughan.

Options

Each instance of the editor has a set of options, which can be set to tailor it to your liking. The command *edit* invokes a version of *ex* designed for more casual or beginning users by changing the default settings of some of these options.

There are three kinds of options: Boolean, numeric, and string options. Options are controlled by the *set* command which can be used to show their current values or to assign new values. The options, their defaults, and a short description of each are given in the following table. A more complete description of each option will be given near the end of this reference manual.

Editor options

Name	Abbr	Default	Description
autoindent	*ai*	*noai*	Automatic indentation
autoprint	*ap*	*ap*	Automatic print after change
beautify	-	*nobeautify*	Discard most non-graphic characters
directory	*dir*	*dir*=/tmp	Directory for editor buffer file
editany	*ea*	*noea*	Allow editing of any file
edited	-	-	Current file is *edited*
errorbells	*eb*	*eb*	Ring terminal bell on errors
fork	-	*fork*	Allow shell escape without write
home	-	*home*=homedir†	Home directory
hush	-	*nohush*	Inhibit all feedback
ignorecase	*ic*	*noic*	Ignore upper/lower case in matching
indicateul	*iu*	*noiu*	Indicate underlining on CRT's
list	-	*nolist*	Print lines (more) unambiguously
magic	-	*magic*‡	More magic characters in regular expressions
de	-	*mode*=644*	Default create mode for files
notify	-	*notify*=5‡	Feedback threshold on changes
number	-	*nonumber*	Number all (input and output) lines
open	-	*open*‡	Allow open and visual commands
optimize	-	*optimize*	Enhance throughput (but lose some typeahead)
printall	*pa*	*printall*	Print all characters
prompt	-	*prompt*	Prompt for input
scroll	-	*scroll*=12	Number of logical lines in a scroll
shell	*sh*	*sh*=/bin/sh	Shell for UNIX escape
shiftwidth	*sw*	*sw*=8	Shift width (tab stop for *autoindent*)
sticky	-	*nosticky*	Post command flags stick around
terse	-	*noterse*	Shorter error diagnostics
ttytype	*tty*	*tty*=unknown†	Terminal type
visualmessage	*vm*	*novm*	Interconsole message inhibition during *visual*
window	-	*window*=23	Window size for z command
wrap	-	*wrap*	Context addressing searches go past top/bottom

† User-dependent (from *htmp* data base)

‡ *Nomagic*, *notify*=1, *noopen* if invoked as *edit*

* Always set and given in octal

Initialization

When it is first invoked, *ex* will use the home directory data base *htmp* to set the *home* directory option and to set the *ttytype* option, reflecting the kind of terminal in use. If there is a file **.exrc** in the user's home directory, then *ex* will *source* to that file. Options setting commands placed there will thus be executed before each editor session.

Entering the editor

Ex is entered by a command of the form

ex [**−**] [**−o**] [**−n**] [**−p**] [[**−r**] *name* ...]

Brackets here indicate optional arguments. The **−** option suppresses all interactive-user feedback and is useful in processing editor scripts in command files. The **−p** option suppresses the *prompt*. The **−n** option is implied by the **−** option and causes the editor to do no **.exrc** or terminal-type dependent start-up processing. The **−o** option causes *ex* to set the terminal type dependent options based on the characteristics of the diagnostic output if the standard output is not a terminal. Finally, the **−r** option is used in recovering after an editor or system crash. See the section on crash recovery below.

File manipulation

Ex is normally editing the contents of a single file, whose name is recorded in the *current* file name. *Ex* performs all editing actions in a buffer (actually a temporary file) into which the text of the file is initially read. Changes made to the buffer have no effect on the file being edited unless and until the buffer contents are written out to the file with a *write* command. After the buffer contents are written, the previous contents of the written file are no longer accessible.

A file argument on the command line causes that file to be initially edited. Its name becomes the current file name, and its contents are read into the buffer.

Edited file notion

Most of the time the current file is considered to be *edited*. This means that the contents of the buffer are logically connected with the current file name, so that writing the current buffer contents onto that file, even if it exists, is a reasonable action. If the current file is not *edited* then *ex* will not normally write on it if it already exists. This protects the user against accidental destruction of files. In all normal editing patterns, the current file is considered *edited*.

Alternate file

Each time a new value is given to the current file, the previous current file is saved as the *alternate* file. Similarly if a file is mentioned but does not become the current file, it is saved as the alternate file. The character '`' substitutes for the alternate file in forming new filenames. This makes it easy to deal alternately with two files and eliminates the need for retyping the name supplied on an *edit* command after a ''No write since last change'' diagnostic is received.

Filename formation

Filenames within the editor may be specified using the normal UNIX expansion conventions: '∗' matches any sequence of characters in a file name, '?' matches any single character, and '[*class*]' matches the set of characters in the class, with single characters specifying themselves, and ranges of the form 'a−z' permitted, this example matching all letters.†

In addition to these metacharacters, the character '%' in filenames is replaced by the *current* file name and the character '`' by the *alternate* file name. If it is necessary for one of the characters '*', '?' '[' '%', '`' or '\' to appear in a filename, it may be escaped by preceding it with a '\'.

---

† Note that an initial character '.' in a filename must always be specified explicitly, as must all '/'s in path names.

Multiple files

If more than one file is given on the command line, then the first file is edited as described above. The remaining arguments are placed with the first file in the *argument list*. The current argument list may be displayed with the *args* command. The next file in the argument list may be edited with the *next* command. The argument list may also be respecified by specifying a list of names to the *next* command. These names are expanded, the resulting list of names becomes the new argument list, and *ex* edits the first file on the list.

Errors

When errors occur *ex* normally rings the terminal bell and prints an error diagnostic. If the primary input is from a file, editor processing will terminate.

Interrupts

If *ex* receives an interrupt signal (ASCII DEL) it prints ''Interrupt'' and returns to its command level. If the primary input is a file, then *ex* will exit when this occurs.

Hangups

If a hangup signal is received and the buffer has been modified since it was last written out *ex* attempts a *preserve* command. If this command fails then *ex* will not unlink the editor buffer in the directory where it was being kept. In either case a *recover* command can be used to continue the work where it left off.

Crash recovery

If the editor or system crashes, or if the phone is hung up accidentally, then you should be able to recover the work you were doing, to within a few (maximum of 15) lines of changes of the place where you were. To recover a file you can use the *recover* command, or the **−r** option, as in

ex −r resume

if you were editing the file *resume*. In order to recover you must have had a current file name when the crash occurred, and respecify this name. After recovering the file you should check that it is indeed ok before writing it over its previous contents. If an error occurs during the recovery operation this means that the buffer was not in a consistent state at the time of the crash and that you will not be able to recover in this way.

Modes

Ex has five distinct modes. The primary mode is *command* mode. Commands are entered in command mode when a ':' prompt is present, and are executed each time a complete line is sent. In *text input* mode *ex* gathers input lines and places them in the file. The *append*, *insert*, and *change* commands use text input mode. No prompt is printed when you are in text input mode. This mode is left by typing a '.' alone at the beginning of a line, or by typing an end-of-file (CTRL(D) at the beginning of a line.)

The last three related modes are *open* and *visual* modes, entered by the commands of the same name, and, within *open* and *visual*, *text insertion* mode. *Open* and *visual* mode allow local editing operations to be performed on the text of a line. *Open* deals with one line at a time on soft-copy terminals while *visual* works on (unintelligent) soft-copy terminals with full-screen addressible cursors. *Visual* uses the entire screen as a (single) window for file editing changes.

Command structure

Most commands have alphabetic names, and initial prefixes of the names are accepted. The ambiguity of short names is resolved in favor of the more commonly used commands, always those of the editor *ed*. Thus the command *print* can be abbreviated 'p' while the shortest available abbreviation for the *preserve* command is 'pre'.

Most commands accept prefix addresses specifying the lines in the file upon which they are to have effect. The forms of these addresses will be discussed below. A number of commands also may take a trailing *count* specifying the number of lines to be involved in the command. Thus the command '10p' will print the tenth line in the buffer while 'delete 5' will delete five lines from the buffer, starting with the current line.

Some commands take other information or parameters, this information always being given after the command name. Examples would be option names in a *set* command i.e. 'set number', a file name in an *edit* command, a regular expression in a *substitute* command, or a target address for a *copy* command, i.e. '1,5 copy 25'.

Feedback

Most commands which change the contents of the editor buffer give feedback if the scope of the change exceeds a threshold given by the *notify* option.† This feedback helps to detect undesirably large changes so that they may be quickly and easily reversed with an *undo*. Thus if a *delete* command eliminates 100 lines you will be informed by a message of the form "100 lines deleted." Similarly, after commands with more global effect such as *global* or *visual*, you will be informed if the net change in the number of lines in the buffer during this command exceeds this threshold.

Command variants

A number of commands have two distinct variants. The variant form of the command is invoked by placing an '!' character after the command name. Some of the default variants may be controlled by options; in this case, the '!' serves to toggle the default. Useful variants are 'quit !' which suppresses warnings about the buffer not having been written out, and 'write !' which allows overwriting of an existing file which is not the edited file.

Command flags

The characters '#' and ':', and the letters 'p' and 'l' may be placed after many commands in any combination. In this case, the command abbreviated by these characters is executed after the command completes. Any number of '+' or '−' characters may also be given with the option flags. If they appear, the specified offset is applied to the current line value before the printing command is executed. The option *autoprint* makes most trailing 'p' characters supplied by *ed* users superfluous; as *autoprint* is suppressed during *global* commands, these flags are still often necessary.

Multiple commands on a line

More than one command may be placed on a line by separating each pair of commands by a '|' character. However the *global* commands, and the shell escape '!' must be the last command on a line, as they are not terminated by a '|'. Thus the command form 'write | next', which can be abbreviated 'w|n', will *write* the current file and then edit the *next* file in the argument list.

---

†Current notable exceptions are *tabulate*, *expand*, and the shift commands '<' and '>'.

Command addressing

As previously mentioned, many commands accept address specifications before the command itself is given. These consist of a series of addressing primitives, described below, separated by ',' or ';'. Such address lists are evaluated left-to-right. When addresses are separated by ';' the current line '.' is set to the value of the previous addressing expression before the next address is interpreted. If more addresses are given than the command requires, then all but the last one or two are ignored. If the command takes two addresses, the first addressed line must precede the second in the buffer. Null address specifications are permitted in a list of addresses, the default in this case is the current line '.'; thus ',$' is equivalent to '.,$'. It is an error to give a prefix address to a command which expects none.

A simple example of command addressing is the command '1,$print' which prints all the lines in the buffer, the first '1' to the last '$'. The command '/^Thesis/;/Example' will search forward to the first line beginning with the string 'Thesis', set the current line to be this line, and then search forward from this line for the string 'Example'. If such a line is found, it is printed.

Addressing primitives

Current and last lines. The current line is refered to symbolically by '.', the last line by '$'. The default address for most commands is the current line, thus '.' is rarely used alone as an address. Most commands leave the current line as the last line which they affect.

Line numbers. The lines in the editing buffer are numbered sequentially from 1; the last line in the buffer may be referred symbolically to as '$'. The most primitive form of addressing refers to lines by their line numbers in the file. Some commands also allow reference to a hypothetical line 0. These commands operate before the first line of the buffer. Thus '0 read header' places a copy of the contents of the file *header* before the first buffer line.

Relative addresses. Addresses may also be specified relative to the current buffer line. Thus '−5' refers to the fifth line preceding the current line while '+5' refers to the fifth line after it. Similarly a single '−' addresses the line before the current line while '++' addresses the second following line. Note that the forms '.+2', '+2' and '++' are all equivalent; if the current line is line 100 they all address line 102.

Context searching. One of the most convenient ways of addressing the lines in the buffer is via ''content addressing'' or ''context searching.'' Here we pick out a pattern in the line we wish to refer to and specify that pattern after the search delimiter '/' to search forwards or '?' to search backwards. If we are simply looking for this pattern, then this is all we need to do; '/Thesis' will search forward in the file and then print the first line, if any, containing the string 'Thesis'. If we wish to give a command to be executed at this line we must close off the search string with a matching delimiter. Thus the command '/Thesis/delete' will delete the next line containing the string 'Thesis'. The pattern here may actually be a regular expression. This allows, e.g. searching for a string at the beginning or end of a line. It is possible to search again for the same pattern by giving a null regular expression; that is either a form such as '//', or a single '/' or '?' immediately followed by a newline character. Context searches normally wrap around past the end of the file if necessary to continue the search.†

Marks. The final way of specifying a line in the buffer is with a *mark*. The *mark* command may be used to give a line a mark, which is denoted by a single lower case letter. Thus 'mark a' will mark the current line with tag *a*. This line may be subsequently referred to in addressing as ''*a*'.

Previous context mark. One mark is automatically set by the editor. This is the previous context mark, referred to in addressing expressions via ''''. Before each non-relative motion of the current line '.', the previous current line is marked with this special tag.‡

---

†It is also possible to use the previous scanning or substitute regular expression for the scan; the forms are '\/' and '\&/' to scan forwards, '\?' and '\&?' to scan backwards respectively.

‡This makes it easy to refer or return to this previous context. Thus if you specify a context search which leads you to a line other than you intended, you may return to the previous current line via ''''.

Command summary

Summarizing the discussion above, the general form of an *ex* command is:

address **command** *! parameters count flags*

All parts are optional; the degenerate case is the empty command which prints the next line in the file.

The following table summarizes *ex* command formats and the shortest allowable abbreviations for commands. Except as noted, all commands which take addresses assume the current line as default if no addresses are given. Each command will be discussed in more detail in the next section.

Command summary

Prototype	Abbr.	Description
append *!*	**a**	Append text after addressed lines
args *!*	**ar**	Print argument list
cd *dir*	**cd**	Synonym for *chdir*
change *!*	**c**	Change text of specified lines
chdir *dir*	**chd**	Change working directory
copy *addr*	**co**	Make a copy of specified lines after *addr*
delete *count*	**d**	Delete specified lines
echo *text*	**ec**	Echo *text* to output
edit *file*	**ed**	Synonym for *ex*
ex *file*	**e**	Edit specified file
expand *count*	**exp**	Expand tabs to spaces
file *file*	**f**	Display/change current file
global */pat/ cmds* †‡	**g**	Execute *cmds* on lines matching *pat*
help *topic*	**h**	Provide information on *topic*
insert *!*	**i**	Insert text before addressed line
join *! count*	**j**	Join lines together
k *x*	**k**	Synonym for *mark*
list *count*	**l**	Print lines more unambiguously
mark *x*	**ma**	Mark addressed line with letter *x*
move *addr*	**m**	Move specified lines after *addr*
next *!*	**n**	Edit next file in argument list
next *! filelist*	**n**	Respecify argument list; edit first file
open */pat/*	**o**	Intraline edit of specified line
preserve	**pre**	Save buffer when disaster strikes
print *count*	**p**	Print addressed lines
put	**pu**	Restore lines
quit *!*	**q**	Terminate editor session
read *file*	**r**	Read *file* into buffer after current line
recover *file*	**rec**	Recover editing buffer after disaster
reset	**res**	Restore option default values
rewind	**rew**	Rewind argument list; edit first file
set *params*	**se**	Set/interrogate options
shell	**sh**	Invoke another, interactive, shell
source *file*	**so**	Read editor commands from *file*
substitute*/pat/repl/ flags count* †	**su**	Substitute *repl* for *pat*
sync	**sy**	Synchronize the temporary file
tabulate *count*	**ta**	Convert (leading) blanks to tabs
transcribe *addr*	**t**	Synonym for *copy*
undo *!*	**u**	Reverse effect of last command
v†	**v**	Synonym for "*global!*" variant
version	**ve**	Print current version information
visual *type*	**vi**	Enter visual mode

Command summary		
Prototype	Abbr.	Description
write *! file* ‡	**w**	Write specified lines to *file*
write *! >> file* ‡	**w**	Write addressed lines at end of *file*
xpand *count*	**x**	Synonym for *expand*
yank *count*	**ya**	Define lines to be *put*
z *type count*	**z**	Context display
! *command*	-	Send *command* to a shell
= ‡	-	Show line number in buffer
> *count*	-	Right shift
< *count*	-	Left shift
EOF	-	Scroll (EOF is generated by CTRL(D))
CR or **NL**	-	Null command prints addressed (next) line
# *count*	-	Synonym for *number*
: *count*	-	Print inhibiting *list* and *number* options.
& *flags count*	-	Repeat last *substitute* command
~ *flags count*	-	Substitute last *repl* for last pattern
|	-	Multiple command per line separator

†*Pat* may be delimited by other characters; '\/' and '\&/' are also permitted as in address formation, and with these forms *repl* is terminated by '/' in a *substitute*.

‡Default address is entire buffer (last line for '=').

Command variants

A number of command have variants, introduced by following the command name with a '!'. These variants are summarized in the following table.

Command variants	
Variant	Description
append *!*	Toggle *autoindent* during *append*
args *!*	Print all arguments, not just those remaining
change *! count*	Like *append!*
ex *! file*	Suppress ''No write'' complaint before executing
edit *! file*	Like *ex!*
global *! /pat/ cmds*	Execute *cmds* on lines not matching *pat*
insert *!*	Like *append!*
join *! count*	Join lines without massaging blank space
next *!*	Like *ex!*
quit *!*	Suppress ''More files'' and ''No write'' complaints
tabulate *! count*	Convert all blanks to tabs, not just initial
undo *!*	No error if ''Nothing to undo'' or ''No change''
write *! file*	Suppress *write* checks (i.e. overwrite file)
write *! >> file*	Like *write!* (*file* can be non-existent)

Command descriptions

In the following command descriptions, the default addresses are shown in parentheses, which are *not*, however, part of the command. The variant flags *!*, *counts* and *flags* are always optional.

(**.**) **append** *!*
text
.

The *append* command reads the input text and places it after the specified line. After the command, '**.**' addresses the last line input or the specified line if no lines were input. If address '0' is given, text is placed at the beginning of the buffer. The variant flag toggles the setting for *autoindent* during the input of *text*.

args *!*

The members of the argument list are given starting with the current one or, if the variant is given, starting with the beginning of the argument list.

cd *directory*

The *cd* command is a synonym for *chdir*.

(**.** , **.**) **change** *! count*
text
.

The *change* command replaces the specified lines with the input *text*. The current line becomes the last line input; if no lines were input it is left as for a *delete*. The variant toggles *autoindent* as in a *append*.

chdir *directory*

The specified *directory* becomes the current directory. If no directory is specified, the current value of the *home* option is used as the target directory. After a *chdir* the current file is not considered to have been *edited* so that write restrictions on pre-existing files apply.

(**.** , **.**) **copy** *addr flags*

A *copy* of the specified lines is placed after *addr*, which may be '0'. The current line '**.**' addresses the last line of the copy. The command *transcribe*, 't', is a synonym for *copy*.

(**.** , **.**) **delete** *count flags*

The *delete* command removes the specified lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end, the new last line becomes the current line.

echo *text*

Text is echoed onto the standard output up to a '|' or newline character. These (and any) characters may be included in *text* by preceding them with a '\'. Initial blanks are stripped from *text*.

edit *! filename*
ex *! filename*

The *edit* command is used to begin an editing session on a new file and is composed of several distinct actions. *Edit* first checks to see if the buffer has been modified since the last *write* command was issued. If it has been, a warning is issued and the *edit* command is never begun. In this case, the user has a second and last chance to write out the buffer. If another *edit* (or *next* or *quit)* command is executed without a *write* and before any further modifications to the buffer, the editing changes to the buffer will be lost. This entire warning procedure is suppressed if the variant flag is given.

The *edit* command next deletes the entire contents of the editor buffer making the named file the current file and printing its name. After insuring that this file is sensible, i.e. that it is not a binary file such as a directory, a block or character special file other than **/dev/tty**, a terminal, or a binary or executable file (as indicated by the first word), *ex* reads the file into the editor buffer.

If the read of the file completes without error, the number of lines and characters read is typed. If there were any dirty (non-ASCII) characters in the file they are stripped of their non-ASCII high bits, and any null characters in the file are discarded. If none of these errors occurred, the file is considered *edited*. If the last line of the input file is missing the trailing newline character, it will be

supplied and a complaint will be issued. This command leaves the current line '**.**' at the last line read.

(**.** , **.**) **expand** *! count flags*

The *expand* command processes the text of the specified lines, converting tabs to an appropriate number of spaces. The current line is left at the last line which had a tab expanded.

file

The current filename is displayed along with an indication of whether it is considered '[Edited]', whether it has been '[Modified]' since the last *write* command, and the number of lines in the buffer.

file *filename*

The current file is changed to *filename* which is not considered *edited*.

(1 , $) **global** *! /pat/ cmds*

The *global* command first marks each line among those specified which matches the given regular expression. Then the given command list is executed with '**.**' initially set to each marked line. In the variant form the list is executed at each line not matching the given regular expression.

The command list consists of the remaining commands on the current input line and may continue to multiple lines by ending all but the last such line with a '\'. *Append*, *insert*, and *change* commands and associated input are permitted; the '**.**' terminating input may be omitted if it would be on the last line of the command list. *Open* and *visual* commands are permitted in the command list and take input from the terminal.

The *global* command itself may not appear in *cmds*. The *undo* command is also not permitted there, as *undo* instead can be used to reverse the entire *global* command. The options *autoprint* and *autoindent* are inhibited during a *global*, and the value of the *notify* option is temporarily infinite, in deference to a *notify* for the entire global. Finally, the context mark '´ ´ ' is set to the value of '.' before the global command begins and is not changed during a global command, except perhaps by an *open* or *visual* within the *global*.

help *topic*

The *help* command accepts keywords related to the editor and, if there is information in its data base about that *topic* supplies the information. A list of topics can be had by *help index*. The data files for help are kept in the directory **/usr/lib/how_ex**.

(**.**) **insert** *!*
text
.

The *insert* command places the given text before the specified line. The current line is left at the last line input; if there were none input it is left at the line before the addressed line. This command differs from *append* only in the placement of text. The variant toggles *autoindent* during the *insert*.

(**.** , **.**+1) **join** *! count flags*

The *join* command places the text from a specified range of lines together on one line. White space is adjusted at each junction to provide at least one blank character. If there is already white space at the end of the line, then the white space at the start of the next line will be discarded. The variant causes a simpler *join* with no white space processing.

(**.**) **k** *x*

The *k* command is a synonym for *mark*.

(**.** , **.**) **list** *count flags*

The *list* command prints the specified lines in a more unambiguous way; non-graphic characters are escaped in octal, tabs and backspaces are printed as ⩾ and ⩽ with the overstruck '−' being omitted if the terminal can not overstrike. The end of each line is marked with a trailing '$'. The current line is left at the last line printed.

(.) **mark** *x*

The *mark* command gives the specified line mark *x*, a single lower case letter. (The *x* must be preceded by a blank or a tab.) Subsequently, the addressing form ‘´ x’ addresses this line. The current line is not affected by this command.

(. , .) **move** *addr*

The *move* command repositions the specified lines after *addr*. The first of the moved lines becomes the current line.

next *!*

The next file from the command line argument list is edited. The variant suppresses ‘‘No write since last change’’ warnings before performing the *next* as for the *edit* command.

next *! filelist*

The specified *filelist* is expanded and the resulting list replaces the current argument list; the first file in the new list is then edited.

(. , .) **number** *count flags*

The *number* command prints each specified line preceded by its buffer line number. The current line is left at the last line printed.

(.) **open** *flags*

(.) **open** /*pat*/ *flags*

The *open* command enters intraline editing mode at each addressed line. If *pat* is given, then the cursor will be placed initially at the beginning of the string matched by the pattern.† Further lines containing *pat* may be opened using the *next* ‘n’ operation without leaving open. The current line is left at the last line opened. See the *open* and *visual* mode description below for more details.

preserve

The current editor buffer is saved as though the editor had just crashed. This command is for use only in emergencies when a *write* command has resulted in an error and you don’t know how to save your work. After a *preserve* you should seek help immediately.

(. , .) **print** *count flags*

The *print* command prints the specified lines with non-printing characters normally escaped as ‘?’. The current line is left at the last line printed.

(.) **put**

The lines removed from the editing buffer by the last command which had the ability to change the buffer are restored after the addressed line. *Put* can be used, e.g., after a *change* command to retrieve the lines changed away when you decide that you want both these and the lines you replaced them with. A *delete* command and a *put* command effect a *move*. Note that *put* is very similar to its *open* and *visual* mode counterpart.

quit *!*

The *quit* command causes *ex* to exit. No automatic write of the editor buffer to a file is performed. However, *ex* issues a warning message if the file has changed since the last write command was issued and it offers a second chance to *write*. *Ex* will also complain if there are more files in the argument list. The variant form suppresses these complaints.

(.) **read** *filename*

The *read* command places a copy of the text of the given file in the editing buffer after the specified line. If no *filename* is given the current file name is used. The current file name is not changed unless there is none in which case *filename* becomes the current name. The sensibility restrictions for the *edit* command apply here also. If the file buffer is empty and there is no current name then *ex* treats this as an *edit* command.

Address ‘0’ is legal for this command and causes the file to be read at the beginning of the buffer. Statistics are given as for the *edit* command when the *read* successfully terminates. After a *read* the current line is the last line read.

---

†The *pat* may be delimited only by ‘/’ characters; the forms ‘\/’ and ‘\&/’ are also not allowed here.

recover *file*

The command *recover* may be used to retrieve the contents of the editor buffer after a system crash, editor crash, or a *preserve* command. A *recover* also occurs implicitly when the **−r** option is specified on the command line. A file name should be given to *recover* unless the file of the current name is to be recovered. Thus a name is always required on the command line. A *recover* results in the removal of the saved buffer. The recovered buffer contents should be checked for sensibility and then saved. It is not possible to recover from errors occurring during a *recover*.

reset

The *reset* command restores the default settings of all numeric and Boolean valued options.

set *parameter*

The *set* command may be used to interrogate and to give new values to options. With no arguments it prints those options whose values have been changed from their defaults; with parameter *all* it prints all of the option values

By giving an option name followed by a '?' the current value of a single option may be interrogated. The '?' is unnecessary unless the option is Boolean valued. Boolean options are given values either by the form 'set option' to set them on or 'set nooption' to set them off; string and numeric options are assigned via the form 'set option=value'. More than one parameter may be given to *set*; they are interpreted left-to-right.

It is also possible to interrogate the current values of the current and alternate file names, and the previous UNIX shell escape command by supplying the parameter '%', '` ', or '!' respectively.

shell

A new *shell* is created. This shell is *interactive*, like a login shell. When it terminates, editing resumes.

source *file*

The *source* command causes *ex* to read commands from the specified file. *Source* commands may be nested.

(. . .) substitute /*pat*/*repl*/ *options count flags*

On each specified line, the first instance of pattern *pat* is replaced by replacement pattern *repl*. If the *global* indicator option character 'g' appears, then all instances are substituted; if the *confirm* indication character 'c' appears, then before each substitution the line to be substituted is typed with the string to be substituted marked with '↑' characters. By typing an 'y' one can cause the substitution to be performed, otherwise no change takes place. After a substitute the current line is the last line substituted.

See the regular expression description for an explanation of metasequences available in *repl*. In addition to these sequences, lines may be split by substituting new-line characters into them. The newline in *repl* must be escaped by preceding it with a '\'. (If the *substitute* is within a *global*, then two escaping '\' characters will be needed.)

sync

The *sync* command causes the contents of the editor temporary file to be synchronized to reflect the current state of editing. *Sync* commands are done automatically whenever there is a difference of 15 lines or more between the in-core buffer and the temporary. They are as useful as *write* commands, and much faster, for those who are worried about losing work due to an editor or system crash.

(. . .) tabulate *! count flags*

The *tabulate* command causes leading white space to be converted to tabs on the specified lines. The variant causes this tabulation to occur throughout each line. The current line is left at the last line where a change occurred.

(. , .) **transcribe** *addr*

The *transcribe* command is a synonym for *copy*.

undo *!*

The *undo* command reverses the changes made in the buffer by the last buffer editing command. Note that *global* commands are considered a single command for the purpose of *undo* (as are *open* and *visual.)* Also, the commands *write* and *edit* which interact with the UNIX file system cannot be undone. *Undo* is its own inverse. After an *undo* the current line is the first line restored or the line before the first line deleted if no lines were restored. For commands with more global effect, such as *global* and *visual* the current line regains it pre-command value after an *undo.* *Undo* always marks the previous value of the current line '**.**' as '´ ´ '.

(1 , $) **v** /*pat* / *cmds*

The *v* command is a synonym for the *global* command variant 'global!'.

version

The *version* command prints the current version number of the editor as well as the date the binary was created.

(**.**) **visual** *type flags*

The *visual* command enters *visual* mode at the specified line. *Type* is optional and may be '+', '−' , '↑' ('^') or '**.**' as in the *z* command to specify the placement of the specified line on the screen. By default, if *type* is omitted, the specified line is placed as the first on the screen. After a *visual*, the current line is the last line the cursor was on when it ended. See the section describing *visual* and *open* for more details.

(1 , $) **write** *! file*

(1 , $) **write** *! >> file*

The *write* command places data from the file buffer back into the file system. The first form of the command will write to a file only if it is the current file and is *edited*, if the file does not exist, or if the file is actually a teletype, **/dev/tty** or **/dev/null**. If the file does not exist it is created. The current file name is changed only if there is no current file name. The current line is unchanged by this command, and feedback is given as to the number of lines and characters written as for the *edit* command. The second form is used to write the buffer contents at the end of an existing file. For both forms, the variant suppresses the file existence and type checks.

If an error occurs while writing the current and *edited* file, *ex* considers that there has been ''No write since last change'' even if the buffer had not previously been modified.

(. , .) **xpand** *count flags*

The *xpand* command is a synonym for *expand*.

(. , .) **yank** *count*

The *yank* command causes the contents of the addressed lines to define the text to be placed in the buffer by a succeeding *put* command. The addressed lines are not affected. A *yank* and a *put* can be used instead of a *copy* command.

(**.**) **z** *type count*

The *z* command gives access to windows of text. The default number of logical lines in a window is given by the numeric *window* option or may be given explicitly by the *count* after the command. The various types and their meanings are:

.	window around the current line
−	window ending at the current line
+	window starting after the current line
omitted	window starting at the current line
↑ or ^	window before this window

In addition, the form *z*= displays a window of text with the current line in the center delimited by lines of '−' characters. For all commands forms except *z*= the current line is left at the last line printed; for *z*= '**.**' addresses the bracketed line.

The characters '+', '^ ' and '−' may be repeated for cumulative effect. On a CRT the screen is cleared before display begins unless a count which is less than the screen size is given.

!*command*

The remainder of the line after the '!' character is sent to a shell to be executed. The current line is unchanged by this command. Within the text of *command* the characters '%' and '` ' are expanded as in filenames and the character '!' is replaced with the text of the previous command. Thus, in particular, '!!' repeats the last such shell escape. If any such expansion is performed, the expanded line will be echoed.

If there has been ''No write'' of the buffer contents since the last change to the editing buffer, then a diagnostic will be produced before the command is executed as a warning. A single '!' is printed when the command completes.

($) =

The '=' command prints the line number of the addressed line. The current line is unchanged.

(. , .) > *count flags*
(. , .) < *count flags*

The '>' right shift and '<' left shift commands perform intelligent shifting on the specified lines. The quantity of shift is determined by the *shiftwidth* option and the repetition of the specification character. Thus '>>' causes the current line to be right shifted two tab stops. Only white space is shifted; no non-white space characters are discarded in a left-shift.† The current line becomes the last line which changed due to the shifting.

EOF

If *ex* receives an end-of-file (control-d) from a terminal input, it interprets this as being a request for a scroll through the file and sends the next *scroll* logical lines of text, normally a half *window*.

(.+1 **)** NL
(.+1 **) |**

An address alone causes the addressed line to be printed. A blank line alone is thus useful for stepping through text.

(. , .) : *count flags*

The *colon* command is used to override the options *number* and *list* so as to print a line without these features while the options remain set.

(. , .) & *options count flags*

The '&' command repeats the previous *substitute* command.

(. , .) ~ *options count flags*

The '~' command replaces the previous regular expression with the previous replacement pattern from a substitution.

---

†White space characters are blank and tab.

Visual and open modes

Ex has two modes, *visual* and *open*, which are quite different from command mode. In command mode, one prepares command lines which are then executed as they are sent to the editor. The editor maintains only a notion of a current line in command mode, not of a current position within that line.

In *open* and *visual* modes, there is both a current line and a current position within that line. The cursor appears on the current line, and indicates the current position within that line by its position. One then forms *operations* consisting of one or more operation characters which are immediately acted upon by the editor. Most operation characters do not show on the screen, rather their effect on the contents of the buffer is shown. Operation sequences do not form ''lines'' of input, and do not need to be terminated by a new-line character.

Operations. Becoming proficient at using *open* or *visual* thus requires learning a set of *operations* which you can use to modify the text of your file. There are a large number of operations. They are associated with keys so as to suggest their function. Related functions are often performed by the upper case counterpart of a lower case operation. Thus the 'f' find operation moves the cursor to the following specified character in the forward direction within the line, while 'F' performs the same function in the backward direction. Similarly, 'r' replaces the character under the cursor with the single following character, while 'R' replaces successive text line characters with the input characters up to a terminating ESC. It is not necessary to learn all or nearly all of the available operations in *open* and *visual*. As you become more proficient with *open* and *visual* you may find use for more of them.

Intraline operations. There are two major kinds of operations in *visual* and *open* − those dealing with the characters of a single line, and those dealing with the lines themselves. The intraline operations deal with the text of a single line. The editor has facilities for referring to the line in terms of a number of characters, a column position, a number of ''words'', a target character, the beginning of the line, the first non-blank character on the line, the end of the line, etc. In addition these operations will take *counts* repeating their effect whenever appropriate. Thus the word move operation 'w' will advance the cursor to the beginning of the next word in the current input line, while the operation '3w' will advance three words.

Interline operations. Interline operations allow the introduction of new lines, the movement of lines, making copies of existing lines, joining together text from several lines to form one new line, substitution of new lines for old, and the deletion of lines. Most of these operations ignore the cursor position within the current line. It is also possible to introduce new lines into the file by inserting text within a line and including a new-line character in this text. This has the effect of ''splitting'' the line into two new lines.

Open mode display. In *open* mode, the text of the current line is displayed with the cursor initially at the first non-blank position of the line. If a regular expression is given following the *open* command then the first character which matched this expression is the character under the cursor. The single displayed logical line usually occupies one physical line on the screen but may, on a cursor-addressible terminal, occupy several lines.†

Visual mode display. In *visual* mode, a number of logical lines are placed on the screen, with long lines folded to occupy several physical lines. The cursor may be moved between these lines and each can be edited as with *open*. In addition, the interline operations listed above may be conveniently performed in *visual* mode.

Empty lines. Physical lines in the display which are not occupied by any portion of logical lines are represented by the character '@' alone at the beginning of the line. Physical lines past the end of file are displayed using the character '~' instead of '@'.

Cleaning up the screen. If you have made a number of line changes, creating empty physical lines displayed as '@', you may wish to maximize the information on the screen. You can do this with the CTRL(Z) operation.

Operation errors. If an operation formation or execution error occurs the terminal bell is rung and any partially formed command is discarded. The bell is also rung when a DELETE is used to cancel an

†On cursor addressible terminals, the second to the last line is used instead of the last line. This avoids screen roll-up problems often associated with the last column of the last line. On terminals without cursor-addressing capability, an error will occur if the line is too long to fit on one physical screen line.

operation and when an ESC is sent when no operation is in progress.

Escape. The ESC escape character is extensively used in *open* and *visual*. As we saw above, it is used to terminate text input. It is also used to abort partially formed commands. Thus '4ESC5x' will delete 5 characters; here we changed our mind after typing a '4' and cancelled it with ESC to start anew with a '5'. If ESC is hit when there is no partially formed command in progress, the editor will ring the bell to let you know that nothing is happening.

Getting out. To get out of *open* or *visual* you should use the 'q' quit operation. If you hit two successive DELETE (i.e. RUBOUT) characters or a single QUIT character, you will also drop out of *open* or *visual*.

Bombing out. If you find a bug in the editor, or if a problem occurs in the system you may find the editor ungracefully terminating either just an *open* or *visual* command or the entire session. In this case, you may be left with the terminal in a funny state so that keys do not echo when you hit them. The thing to note in this situation is that your normal erase character and kill line sequences will, most likely, not work, and also that the carriage return character may be different from the new-line character, the former being, quite often, non-functional. The way to recover from this is to type the command:†

stty cooked echo −nl

If you are still talking to the editor you will have to put the escape '!' in front of this command. Note especially that you must type this command without mistakes, and that you must terminate it with a newline. (This entire operation may be difficult because you won't be getting any echo from the system.) If you make a mistake, just send the mangled line with a newline and start over. If you stick a single, unmatched ''' on the end of the line, the shell will scream about the syntax error and not execute the garbage you typed.

Sync. If you have made a number of changes and wish to insure them against a system crash, you can invoke the *sync* command from within *visual* or open via the CTRL(S) operation.

---

†If you are lucky, your system may have the *tset* command which performs this function without requiring any arguments. Then all you will have to do is type 'tset' followed by a new line character, possibly preceded by a '!' if the editor is still with you.

Operation descriptions

Scope of operation. All changes in *open* or *visual* are limited in scope to the visible screen text. Each single change may be reversed with the undo 'u' operation. In addition, a disastrous *open* or *visual* command may be completely undone at the command level.

Format of the operations. Most operations take an optional preceding count, given as a decimal number (not starting with a digit '0'). A number of operations take a following *text* string which is inserted into the buffer as specified by the operation. Some operations take a following operation, called the targeting operation, to indicate the scope on which they are to have effect. This second ''targeting'' operation specifies the cursor motion for the first operation. Thus a simple operation would be 'x' deleting the (single) character under the cursor. We could delete two characters by specifying '2x' or 'xx', the former being preferred. An example of an operation which takes targeting is delete 'd', thus 'dw' will delete a word. Finally, the insert operation is typified by 'ifooESC' where here the text 'foo' is inserted before the current cursor position. The character ESC here is used to terminate the text input.

Definition of "word". There are two different definitions of ''word'' used in *open* and *visual*. The primary definition is a sequence of letters, digits, and underscores, or a sequence of other (non-white) characters followed by trailing white space (blanks and tabs). This is the conservative definition of word. The other, liberal, definition of a ''word'' treats it simply as a maximal sequence of non-blanks with trailing white space. There are two sets of word operations; 'w' and 'b' are conservative, 'W' and 'B' liberal. The back word CTRL(W) operation in text insert mode is liberal; it is especially useful for fast typists who want to quickly and accurately back over several mangled input words.

Intraline motion operations. There are four basic kinds of intraline motion operations − those dealing with characters, those dealing with ''words'', those dealing with targets (either single characters or specified column positions), and, finally, special motions e.g. to the beginning of, first non-blank character of, or end of a line. The basic character oriented operations are SPACE advancing one position to the right and CTRL(H) a backspace which backs up to the right.† The basic word oriented operations are 'w' moving forward to the beginning of the next word, and 'b' moving backward to the beginning of the preceding word.

Single character targets. The character targeting operations are 'f', 'F', 't' and 'T'. Each takes a single following character and searches the current line for that character. The lower case operations search to the right, the upper case operations to the left. The 'f' and 'F' (find) operations are inclusive; that is, they reference through to the specified characters. The 't' and 'T' (to) operations are not inclusive but rather move the cursor up to the specified target.

Special motions. The special intraline motion sequences are '^ ' specifying the first non-blank character on the line; '$' specifying the last character on the line; and '0' specifying the first position on the line. Finally, there is an operation '|', used with a preceding count, which references the column position specified by the count, much as a 'f' operation would.

Operators and targeting operations. Some operations are actually prefix operators, taking another operation, called the targeting operation or *target* after them to indicate the scope on which they are to have effect. There are four such operations − 'c' change, 'd' delete, 'g' grab, and 'y' yank. The first two are by far the most important. The targeting operation must be a intraline motion sequence. Thus 'c2w' could begin an operation changing the next two words in the current line, while 'dt)' would delete the text up to the next ')' character in the current line.

Interline motions. The most primitive interline motion operations are those which advance integral numbers of lines, typified by a carriage return CR or new line NL. There are two kinds of such operations in each direction − the pure cursor motion operations which maintain the current column position as much as possible, and the motion sequences which advance to the first non-blank position of the target line.

Interline motions which respect the current column position include 'k' (also CTRL(K) on an ADM-3A) moving up a line, and NL or 'j' or CTRL(J) moving down a line. Motion sequences which place the cursor in

†On the ADM-3A the control functions of the keys 'h', 'j', 'k', and 'l' perform the left, down, up, and right cursor motions respectively. Hence, for convenience on this most commonly available terminal, the operations 'h', 'j', 'k' and 'l' perform as their control-key counterparts in repositioning the cursor. Thus, in the present cases, 'h' works as well as CTRL(H), and 'l' is equivalent to SPACE.

the first non-blank position are '+' or CR moving down, and '−' moving up. These motion sequences take counts thus '5−' will move back five lines. There are also special sequences 'H' for home which returns the cursor to the first non-blank character of the first line on the screen, and 'L' to the first non-blank character of the last.

Insertion. Text insertion is indicated by *text* in the operation descriptions below. Pure text insertion is begun with the 'i' or 'a' operations and continues to an ESC. If the first character of *text* is the null character, generated by a CTRL(@), then the previous inserted text is re-used. *Text* may contain new-line characters which cause the current line to be split and a new line to be added to the buffer. A number of control characters may be used to edit the inserted text while inserting. These include CTRL(H) to back over a character, CTRL(W) to back over a word (liberal definition), '@' to delete the (current line portion) of the input. Also, the character CTRL(X) is interchangeable with '@' here and as an deleting operation. To enter any of these special characters into the input line they must be preceded by a '\'. This applies also to the DELETE, QUIT and CTRL(D) characters. The first two normally cause termination of the text insert; CTRL(D) is used as a backtab in *autoindent* and otherwise normally ignored.

Convenient intraline insertion abbreviations are 'I' adding text before the first non-blank of the current line, and 'A' adding text at the end of the line. These are similar to the two character sequences '^ i' and '$a' respectively.

Deletion. The deletion operator 'd' may be placed before any intraline motion sequence to form a deleting operation, deleting the moved over text from the current line. Thus 'dw' will delete a word while 'd40|' will delete to column 40. Convenient deletion abbreviations are 'x' deleting characters, 'X' deleting the specified number of preceding characters, and '#', similar to 'X' except that it deletes the character at the cursor while 'X' deletes the character before the cursor. Finally there are the abbreviations 'D' which deletes to the end of the line, i.e. 'd$', and '@' which deletes to the beginning of the line, i.e. 'd0'. Note also that the operation CTRL(X) is a synonym for '@'.

Change. Similar to the delete operations are the change operations, which are formed with 'c' and any motion sequence. The specified text is deleted, indicated by marking the right end of it with a '$' character, and then the input up to an ESC replaces it. Thus 'cwfooESC' will replace the current word with the word 'foo'. Useful abbreviated changes are 'C' which changes the rest of the line, i.e. 'c$', and 's' which changes the number of characters specified by the preceding count.

Replace. There are two forms of the replace operation. The first, 'r', replaces the single character under the cursor with the single following character (no terminating ESC is required.) The second form 'R' replaces as many following characters as are typed with the new input characters. This operation is essentially an ''overstrike'' much as a normal terminal display functions. It is useful in editing pictures and other data where a fixed field size is to be maintained.

Interline inserts. Some operations add a new line to the buffer. These ignore the cursor position on the current line, and do not split it, rather creating a new line. The two operations of this type are 'o' which adds a new line after the current line and 'O' which adds a new line before the current line. In both cases, following text up to an ESC defines one or more new lines. A count may be usefully given before 'o' or 'O' indicating the number of physical lines to be opened up. If this is an estimate of the number of lines to be added it can help to minimize the output required to redraw the screen on terminals which are unintelligent. Thus an appropriate beginning of a sequence to add three new lines after the current line would be '3o'.

Line deletes, joins. Lines may be deleted from the buffer using the command '\\' (two backslashes) or joined together using 'J'. A specified number of lines may be replaced with new text conveniently using the line substitute operation 'S'.

Interline scans. It is possible to scan between lines for text specified by a regular expression and, in *visual*, to specify where this line is to appear on the screen if it must be redrawn. Forward scans are begun with '/' and backward scans with '?'. After hitting '/' or '?' you enter the pattern you wish to scan for and it is shown on the bottom line of the screen. You can terminate the pattern either with an ESC or a CR or NL; to abort a partially formed scanning operation you can type a DELETE or RUBOUT character. If the search fails the bell is rung, the scan delimiter '/' or '?' is replaced with a 'F' indicating a failed search, and the cursor returns to its previous position with typeahead discarded. If the search succeeds then the cursor is placed at the beginning of the string which matched. The screen is redrawn with the line matched in the

center unless (in *visual* only) a specific positioning request has been made by following the pattern with 'z' or 'v' and then one of '↑', '-', '.', or a CR or NL specifing the top of the screen.

Scrolling. The operation CTRL(D) may be used, as in command mode, to effect a scroll. The number of lines to be scrolled may be specified by a preceding count; this count will hold for succeeding *open* and *visual* scrolls until respecified.

Context displays. Sequences 'z*type*' and 'v*type*' may be equivalently used to specify context display as in command mode. In *open* mode the *type* is not required and a command mode like 'z=' command is always done. In *visual type* may be any of '↑', '-', '**.**' or CR or NL specifying the top of the screen.

Memory. The editor remembers the last *visual* or *open* command and associated data in each of several categories and allows it to be respecified by a very short, one character sequence.

Last single character scan. The last of the targeting operations *f*, *F*, *t*, and *T* is remembered with the character supplied to it. This combination is used again through the operation ';'.

Last interline scan. The last of the interline scans using '/' or '?' is remembered and may be repeated with the operation *n* (next.)

Last modifying command. The last command which modified the buffer is remembered and may be repeated by the command form '**.**'.

Last inserted text. The text which was last inserted (up to 128 characters) is remembered and may be specified in future operation by a null character, generated by a CTRL(@). This is given when *text* would begin, instead of *text.* The ESC terminating the *text* is not needed. If there is no previous inserted text, or if the previous inserted text was longer than 128 characters, the bell is rung and the operation completes inserting no text. If the aborted operation was a scan via '/' or '?', then it aborts as though it had been cancelled with a DELETE character.

Last deleted text. The last deleted text which was part of a single line (up to 128 characters) is remembered. If the last thing deleted was one or more lines, then this will be remembered instead. There are put operations 'p' and 'P' which allow this deleted text to be returned to the buffer. Note that a number of operations set both the deleted and inserted text (notably change operations.)

Grab, yank, and put. There are two related operators yank 'y' and grab 'g' which take a motion sequence target and pretend it was the previous inserted or deleted text respectively. The grab 'g' operation is especially useful when you wish to search for something on the screen − you can grab it, e.g. if it is a word with 'gw', and then do a scan defaulting the search with a null character, i.e.: '/CTRL(@)'. There are also operations 'p' and 'P' which put text which was deleted back into the buffer. If the previous deletion was of lines, then these operations will add new lines with the same text after or before the current line respectively. Similarly if the previous deletion was a part of a line, then the text will be put after/before the cursor position in the current line. There is also an operation 'Y' which yanks a specified number of lines as though they had been deleted but does not delete them. This can be used to copy lines. As an example, the sequence 'Yp' places a copy of the current line after the current line.

Interline motions. The are a number of interline motion sequences dealing with the mark registers and specific line numbers. The operation 'G' causes the line specified by the preceding count to become the current line. If this line is on the screen, then the screen is not redrawn. The default line for 'G' if no count is given is the last line of the file. Thus 'G' is the easiest way to get to the end of the file.

The sequences '´ *x*' where *x* is a single lower case letter cause the display to return to the specified marked line, with the marked line in the center. The previous context mark '´ ´ ' may also be requested here, and it is set by the searching operations '/' and '?', the 'G' operation, the mark operations '´ *x*', and the 'v' or 'z' operations when a count is given. Marks may be set while in *visual* or *open* by using the *K* operation and following it by a single lower case letter specifying the register to be marked.

Miscellaneous notes on visual and open

The options *beautify* and *indicateul* are suppressed in *open* and *visual*. All the features of *autoindent* are available. If the cursor is at a tab character in the line which is represented by a number of blanks, it is placed at the last blank. Lines yanked with 'Y' or deleted with '\\' may be put with 'p' or 'P' in a later

visual or *open* command only so long as no *edit* or *next* command intervenes. It is not possible to *undo* an appending operation in *open* or *visual* which resulted in the creation of more than a screen full of lines. An operation affecting only the text within a single line is undoable only while the cursor remains on that line.

Visual and open mode summary

The following table summarizes the *visual* and *open* operations. For each operation we indicate its general form, whether it can take a *count*, and whether it can be used as a *targeting* operation.

Open and visual operations			
Operation	Count?	Target?	Description
a*text* ESC	yes	no	Append *text* after cursor
b	yes	yes	Backwards words
c*target text* ESC	no	no	Change *target* to *text*
d*target*	yes	no	Delete *target*
e	yes	yes	To end of word (unimplemented)
f*char*	yes	yes	Find *char* to right of cursor
g*target*	no	no	Defi ne previous inserted text
h (←)	yes	yes	Backwards characters (like CTRL(H))
i*text* ESC	yes	no	Insert *text* before cursor
j (↓)	yes	no	Cursor down lines, same column if possible
k (↑)	yes	no	Cursor up lines, same column
l (→)	yes	yes	Forwards character
n	no	no	To next line matching the previous *scanning* regular expression (as described below) in the direction of the previous *open* or *visual* intraline search using '/' or '?'.
o*text* ESC	yes	no	Open a new line after the current line leaving room for the specifi ed number of physical lines. Enter text insert mode on that line.
p	no	no	Put the text lines last deleted with '\\' or the last yanked 'Y' lines after the current line. If the last deletion was of part of a line rather than a 'Y' or '\\' then that deleted text is place in the current line after the cursor.
q	no	no	Quit *open* or *visual*, returning to command mode. The command level *undo* can reverse the entire *open* or *visual* command.
r*char*	yes	no	Replace each of the specifi ed number of characters with a *char*
s*text* ESC	yes	no	Replace the specifi ed number of characters with the specifi ed *text*
t*char*	yes	yes	Cursor right to just before *char*
u	no	no	Undo last change
v	no	no	(In *open*) do command mode 'z=' returning to *open* mode on current line.
v*spec*	yes	no	(In *visual*) with *spec* one of '.', '−', '↑' or '^ ', '+', or CR or NL does the specifi ed type of *visual* command at the specifi ed line, defaulting to the current line
w	yes	yes	Forward to beginning of each word
x	yes	no	Delete characters
y*target*	no	no	Defi ne previous deleted text
z	-	-	Synonym for *v*
A*str* ESC	yes	no	Append at end (short for '$a')
B	yes	yes	Back word (simple blank/non-blank)

Open and visual operations			
Operation	Count?	Target?	Description
C*str* ESC	no	no	Change to end (short for ‘c$’)
D	no	no	Delete to end (short for ‘d$’)
E	yes	yes	Back to end of previous word (unimplemented)
F*char*	yes	yes	Find *char* to left of cursor
G	yes	no	Goto specified line; last line default
H	no	no	To first non-blank on first screen line
I	yes	no	Insert before first non-blank character (i.e. ‘^ i’)
J	yes	no	Join lines
K*x*	no	no	Mark current line in mark register *x*
L	no	no	To first non-blank character on last screen line
O*text* ESC	yes	no	Like *o* but before current line
P	no	no	Like *p* but before current line or before cursor
R*text* ESC	no	no	Replace (overstrike) with input *text*
S*text* ESC	yes	no	Replace specified number of lines
T	yes	yes	Like *t* but scanning to left of cursor
W	yes	yes	Forward word (simple blank/non-blank)
X	yes	no	Delete preceding characters
Y	yes	no	Yank lines, copying them without deleting them so that they may be put with *p* or *P*.
SPACE	yes	yes	Right one character
0	no	yes	To first character of line
↑ or ^	no	yes	To first non-white character
$	no	yes	To end-of-line
@	no	no	Delete characters before cursor
#	yes	no	Delete characters backwards, starting with the character under the cursor
.	no	no	Repeat last modifying command
;	yes	yes	Repeat last *f*, *F*, *t*, or *T* operation
\\	yes	no	Delete lines
+ or CR	yes	no	Forward lines to first non-blank
−	yes	no	Backwards lines to first non-blank
/*re* ESC	no	no	Forward to first line matching *re*. To cancel the search, send a DELETE or RUBOUT.
?*re* ESC	no	no	To previous line matching *re*
/*re* /z*type* ESC	no	no	(In visual) performs the specified type of a *z* or *v* command before the target specified with /*re*/ or ?*re*?.
|	yes	yes	To specified column or column before last up/down line movement
CTRL(D)	yes	no	Down *scroll* lines; in *visual* hold the cursor’s relative position on the screen. If a count is given it becomes the number of logical lines to scroll in *open* or *visual* until another such count is given.
CTRL(S)	no	no	Do a *sync* command
CTRL(W)	yes	yes	Synonym for ‘B’

Open and visual operations			
Operation	Count?	Target?	Description
CTRL(X)	no	no	Synonym for '@'
CTRL(Z)	no	no	Maximize information on screen (clean-up)
ESC	-	-	Cancel partially formed command
RUB	-	-	Cancel a partially formed command. If repeated, drop out to *command* mode
QUIT	-	-	Drop out to *command* mode

Text insertion mode corrections

The following sequences are used in making corrections to text being added in *text insertion* mode. They are also used when entering the regular expression *re* for a interline search using '/' or '?'.

Text insertion mode editing sequences	
Sequence	Action
CTRL(H)	Back a character
@	Delete all input on current line
CTRL(X)	Synonym for '@'
CTRL(W)	Delete a word (simple blank/non-blank definition)
RUB	Drop out of text insert, and also *visual* or *open*
QUIT	Like RUB
special	With special any of the above chars, gives *special*
CR or NL	End current line, rest of text to a new, following line
ESC	Terminate the *text*

Substitute replacement patterns

There are several metacharacters which may be used in substitute replacement patterns. As is the case for the regular expression metacharacters, there are fewer replacement pattern metacharacters if *nomagic* is set. This is discussed more below. In fact, with *nomagic* the only replacement pattern metacharacter is the escaping '\' (this is the default for *edit*).

The basic metacharacters for the replacement pattern are '&' and '~'. These are given as '\&' and '\~' when *nomagic* is set. The metacharacter '&' is by far the most important of these. Each instance of this metacharacter is replaced by the characters which the regular expression matched. Thus the substitute command

substitute/some/& other/

will replace the string 'some' with the string 'some other' the first time it occurs on the current line. The metacharacter '~' stands, in the replacement pattern, as it did in regular expression formation, for the defining text of the previous replacement pattern.

Other metasequences are possible in the replacement pattern, and are introduced by the escaping character '\'; this is the default for *edit*. The sequence '*n*' is replaced by the text matched by the *n*-th regular subexpression enclosed between '\(' and '\)'.† The metasequences '\u', '\l', '\U', '\L', and '\E' and '\e' are used to perform systematic case conversion of letters. The sequences '\u' and '\l' cause the immediately following character in the replacement to be converted to upper- or lower-case respectively if this character is a letter. The sequences '\U' and '\L' turn such conversion on, either until '\E' or '\e' is encountered, or until the end of the replacement pattern. By bracketing selected portions of a regular expression with '\(' and '\)' and using '\U' or '\L' it is possible to systematically capitalize entire words or phrases.

Regular expressions

Ex supports a form of regular expression notation. A regular expression specifies a set of strings of characters. A member of this set of strings is said to be *matched* by the regular expression. Regular expressions may be used in locating or selecting lines by their content, in *open* and *visual* modes to position the cursor within the file, and in the *substitute* command to select the portion of a line to be substituted.

Ex remembers two previous regular expressions: the previous regular expression used in a *substitute* command and the previous regular expression used elsewhere (referred to as the previous *scanning* regular expression.) The previous regular expression can always be referred to by a null *re*, e.g. '//' or '??'.

Magic and nomagic

The regular expressions allowed by *ex* are constructed in one of two ways depending on the setting of the *magic* option. The *ex* default setting of *magic* gives quick access to a powerful set of regular expression metacharacters. The disadvantage of *magic* is that the user must remember that these metacharacters are *magic* and precede them with the character '\' to use them as ''ordinary'' characters. With *nomagic*, the default for *edit*, regular expressions are much simpler, there being only two metacharacters. The power of the other metacharacters is still available by preceding the (now) ordinary character with a '\'. Note that '\' is thus always a metacharacter.

The remainder of the discussion of regular expressions assumes that that the setting of this option is *magic*. To discern what is true with *nomagic* it suffices to remember that the only special characters in this case will be '↑' at the beginning of a regular expression, '$' at the end of a regular expression, and '\'.‡

Basic regular expression summary

The following basic constructs are used to construct *magic* mode regular expressions.

---

†When nested, parenthesized subexpressions are present, *n* is determined by counting occurrences of '\(' starting from the left.

‡With *nomagic* the characters '~' and '&' also lose their special meanings related to the replacement pattern of a substitute.

Basic regular expression forms	
Form	Meaning
char	An ordinary character which matches itself. The character '↑' ('^ ') at the beginning of a line, '$' at the end of line, '∗' as any character other than the first, '.', '\', '[', and '~ ' are not ordinary characters and must be escaped (preceded) by '\' to be treated as such.
↑	Up-arrow (or circumflex '^ ') at the beginning of a pattern forces the match to succeed only at the beginning of a line.
$	At the end of a regular expression forces the match to succeed only at the end of the line.
.	A period character matches any single character except the new-line character.
\<	This sequence in a regular expression forces the match to occur only at the beginning of a ''variable'' or ''word''; that is, either at the beginning of a line, or just before a letter, digit, or underline and after a character not one of these.
\>	Similar to '\<', but matching the end of a ''variable'' or ''word'', i.e. either the end of the line or before character which is neither a letter, nor a digit, nor the underline character.
[*string*]	A string of characters enclosed in square brackets matches any (single) character in the class defined by *string*. Most characters in *string* define themselves. A pair of characters separated by '−' in *string* defines the set of characters collating between the specified lower and upper bounds, thus '[a−z]' as a regular expression matches any (single) lower-case letter. If the first character of *string* is an '↑' or '^ ' then the construct matches those characters which it otherwise would not; thus '[^ a−z]' matches anything but a lower-case letter (and of course a newline). To place any of the characters '↑', '^ ', '[', or '−' in *string* you may escape them by preceding them with a '\'.

More complicated regular expressions are built by putting these simple pieces together. The concatenation of two regular expressions matches the longest string which can be divided with the first piece matching the first regular expression and the second piece matching the second. Thus the regular expression '**..**e' will match any three characters ending in the character 'e', while '^ [aeiou]' matches any vowel which appears at the beginning of a line.

Any of the (single character matching) regular expressions mentioned above may be followed by the character '∗' to form a regular expression which matches any number of adjacent occurrences (including 0) of characters matched by the regular expression it follows. The character '~ ' may be used in a regular expression, and matches the text which defined the replacement part of the last *substitute* command. A regular expression may be enclosed between the sequences '\(' and '\)' with side effects in the *substitute* command, and an escaped digit, e.g. '\1', matches the text which was matched by the corresponding previous '\(' and '\)' bracketed expression, numbered in order of occurrence of the '\(' delimiters.

Option descriptions

autoindent, ai — default: noai

The *autoindent* option can be used to ease the preparation of structured program text. At the beginning of each *append*, *change* or *insert* command or when a new line is *opened* or created by an *append*, *change*, *insert*, or *substitute* operation within *open* or *visual* mode, *ex* looks at the line being appended after, the first line *changed* or the line inserted before and calculates the amount of white space at the start of the line. It then aligns the cursor at the level of indentation so determined.

If the user then types lines of text in, they will continue to be justified at the displayed indenting level. If more white space is typed at the beginning of a line, the following line will start aligned with the first non-white character of the previous line. To back the cursor up to the preceding tab stop one can hit CTRL(D). The tab stops going backwards are defined at multiples of the *shiftwidth* option. You **cannot** backspace over the indent, except by sending an end-of-file with a CTRL(D).

Specially processed in this mode is a line with no characters added to it, which turns into a completely blank line (the white space provided for the *autoindent* is discarded.) Also specially processed in this mode are lines beginning with an '^ ' or '↑' and immediately followed by a CTRL(D). This causes the input to be repositioned at the beginning of the line, but retaining the previous indent for the next line. Similarly, a '0' followed by a CTRL(D) repositions at the beginning but without retaining the previous indent.

Autoindent doesn't happen in *global* commands or when the input is not a terminal.

autoprint, ap — default: ap

The *autoprint* option causes the current line to be printed after each *delete*, *copy*, *join*, *move*, *substitute*, *tabulate*, *transcribe*, *undo*, *xpand* or shift command. This has the same effect as supplying a trailing 'p' to each such command. *Autoprint* is suppressed in globals, and only applies to the last of many commands on a line.

beautify — default: nobeautify

Causes all control characters except tab, newline and form-feed to be discarded from the input. A complaint is registered the first time a backspace character is discarded. *Beautify* does not apply to command input, or to text insertion mode. It applies only when you have entered text input mode by issuing a *insert*, *delete*, or *change* command from command mode.

directory, dir — default: dir=/tmp

The *directory* option specifies the directory in which *ex* places it buffer file. If this directory in not writable, then the editor will exit abruptly when it fails to be able to create its buffer there.

edited — no default

The current file is considered to be *edited* when the buffer contents are directly related to it. In this case the *write* command will write to the file even though it exists. In all normal editing patterns the current file is considered *edited*.

When the current file name is explicitly changed by the *file* command, then the file is not considered *edited* to protect a previous existing file of the same name from accidental destruction.

If a file is not successfully read in by an *edit* command, then it is not considered *edited* so that the possibly incomplete image of the file in the editing buffer will not be accidentally written over its contents.

editany, ea — default: noea

Disables the *edit* and *read* command file sensibility checks.

errorbells, eb — default: eb

If *eb* then error messages are preceded by two bells. The bell ringing in *open* and *visual* on errors is not suppressed by setting *noeb*.

fork default: fork

If *nofork* shell escapes will be inhibited the first time they are attempted if there has been ''No write'' of the buffer since the last change occurred. In this case, the aborted command can be repeated by using the command form '!!'. If *fork*, the default, a warning is given, but the command is given to a shell for execution anyways.

home default: user-dependent

The *home* directory is an image of the user's entry in the *htmp* data base. It is used initially as the origin of the file **.exrc** and is the default directory for the *chdir* command.

hush default: nohush

Inhibits interactive diagnostic information including prompts, printing of file names, line and character counts, command feedback, and echoing by the '!' shell escape.

ignorecase, ic default: noic

If *ignorecase* is set, all upper case characters in the text are mapped to lower case in regular expression matching. In addition, all upper case characters in regular expressions are mapped to lower case except in character class specifications.

indicateul, iu default: noiu

If *indicateul* is set, non-blank characters overstruck with underlines (and vice-versa) cause output lines to be split into two parts for printing − the text and the underlining.

list default: nolist

If *list* is set, all printed lines will be displayed (more) unambiguously, as is done by the *list* command.

magic default: magic†

If *nomagic* is set, the number of regular expression metacharacters is greatly reduced, with only '↑' or '^ ' and '$' having magic effects. In addition the metacharacters '~ ' and '&' of the replacement pattern are treated as normal characters. All the normal metacharacters may be made *magic* when *nomagic* is set by preceding them with a '\'.

mode default: mode=644

Mode is the value the permission bits of any file created by the *write* command will have initially. The default allows reading and writing of the created file by its owner, as well as reading of the file by others.

notify default: notify=5‡

The *notify* option specifies a threshold for feedback from commands. Any command which modifies more than the specified number of lines will provide feedback as to the scope of its changes. For commands such as *global*, *open*, *undo*, and *visual* which have potentially more far reaching scope, the net change in the number of lines in the buffer is presented at the end of the command, subject to this same threshold. Thus notification is suppressed during a *global* command on the individual commands performed.

number default: nonumber

The *number* option may be set to cause all output lines to be printed with their line numbers. In addition each input line will be prompted for by supplying the line number it will have.

open default: open†

If *noopen* then the commands *open* and *visual* are not permitted. This is set for *edit* to prevent confusion resulting from accidental entry to *open* or *visual* mode.

---

†Default is *nomagic* for *edit*.

‡*Notify*=1 for *edit*.

†*Noopen* for *edit*.

optimize default: optimize

If *optimize* is set, and the terminal permits, throughput of text is expedited by setting the terminal to not do automatic carriage returns and to *noecho* mode when printing more than one (logical) line of output. This can greatly speed output on terminals without addressible cursors if leading blanks are present, but causes all type-ahead to be lost. With *optimize*, after a multi-line print you cannot type ahead until the prompt reappears.

printall, pa default: noprintall

If *printall* is set then all characters will be printed in the output as they appear in the line when the *print* command is used. Normally, the blanks and tabs in the line may be rearranged or replaced with cursor addressing to improve throughput, and non-printing characters are printed as '?'. With *printall* these modifications to the pure line text are suppressed.

prompt default: prompt

Command mode input is prompted for with a ':' if the *prompt* option is set.

scroll default: scroll=12

The *scroll* option determines the number of logical lines scrolled when an end-of-file is received from a terminal input.

shell, sh default: sh=/bin/sh

The *shell* option gives the path name of the shell forked for the shell escape command '!'.

shiftwidth, sw default: sw=8

The *shiftwidth* option gives the width a software tab stop. This is used in reverse tabbing with CTRL(D) when using *autoindent* to append text, and by the shift commands. Note that the tab character always references tab stops every eight positions and is **not** affected by this option.

sticky default: nosticky

If *sticky* is set then flags like 'l', '#' and ':', given after a command, stick around until new flags are specified, allowing more lines to be printed in the same way without permanently changing options or continually hitting these keys.

terse default: noterse

If *terse* is set, shorter error diagnostics are produced. These are designed for the experienced user.

ttytype, tty default: terminal dependent

This option gives the terminal type of the output device. Setting *ttytype* indicates the special capabilities of the terminal in use. Current terminal types are defined by the *ttycap* data base. This data base is conveniently handled using the *ttytype* program. For most hard-wired ports, *ttytype* should be set correctly automatically. A mechanism exists using *ttytype* (VI) for setting the type semi-automatically on dial-up lines.

visualmessage, vm default: novm

Interconsole messages are prohibited during *visual* commands if *novm* is set.

window default: window=23

The number of lines in a text window for the *z* command is specified by the *window* option.

wrap default: wrap

If *nowrap* then searches using the regular expressions in addressing will not wrap around past the end of the file.

Limits

The editor limits that the user is likely to encounter are as follows: 512 characters per line, 256 characters per global command list, 64 characters per file name, 128 characters in the previous inserted and deleted text in *open* or *visual*, 100 characters in a shell escape command, 30 characters in a string valued option, and 256K characters in the temporary file.† The limit on the number of lines depends on the amount of core: each line takes at least 1 word, and to effect *undo* occasionally up to 2. This editor is much larger

†Repeated blanks and tabs are compressed in the temporary file, as are blanks between "words". This makes the buffer effectively larger than 256K characters; if the file being edited contains many repeated blanks, it may be editable even if it is larger than 256K.

than *ed* and on PDP 11/40's which do not have separate instruction and data space it is limited to about 2000 lines if *visual* or *open* are ever used. If a full core load of user space is not available *ex* may not be usable. On a PDP 11/45 or 11/70 the size of the editor is not a problem as it can run with separate instruction and data.

Notes on temporary file overflow

This editor uses a temporary file as a workspace. The management of this file is done in the same way as in *ed*. Each line in the file is represented by an in-core pointer to the image of that line on the disk.

The important point to note here is that the editor does *not* reclaim space in this temporary file used by lines which are deleted or changed. This means that files which are larger than 128K characters may be difficult to edit. Similarly systematic changes on large numbers of lines may run the editor out of temporary file space.

If the editor runs out of temporary space you can write the file and then use an *edit* command to read it back in. This will reclaim the lost space. A better solution is to split the file into smaller pieces or to use a stream editor such as *gres* on the file. *Gres* is described in section I of the *UNIX Programmers Manual.*