



# Credits and Acknowledgements

Material in this *Editing and Text Processing on the Sun Workstation* comes from a number of sources: *An Introduction to Display Editing with Vi*, William Joy, University of California, Berkeley, revised by Mark Horton; *Vi Command and Function Reference*, Alan P. W. Hewett, revised by Mark Horton; *Ex Reference Manual*, William Joy, revised by Mark Horton, University of California, Berkeley; *Edit: A Tutorial*, Ricki Blau, James Joyce, University of California, Berkeley; *A Tutorial Introduction to the UNIX Text Editor*, Brian W. Kernighan, Bell Laboratories, Murray Hill, New Jersey; *Advanced Editing on UNIX*, Brian W. Kernighan, Bell Laboratories, Murray Hill, New Jersey; *Introducing the UNIX System*, Henry McGilton, Rachel Morgan, McGraw-Hill Book Company, 1983. These materials are gratefully acknowledged.

**Sun Microsystems, Sun Workstation**, and the combination of **Sun** with a numeric suffix are trademarks of Sun Microsystems, Inc.

**UNIX, UNIX/32V, UNIX System III, and UNIX System V** are trademarks of AT&T Bell Laboratories.

Copyright © 1986 by Sun Microsystems Inc.

This publication is protected by Federal Copyright Law, with all rights reserved. No part of this publication may be reproduced, stored in a retrieval system, translated, transcribed, or transmitted, in any form, or by any means manual, electric, electronic, electro-magnetic, mechanical, chemical, optical, or otherwise, without prior explicit written permission from Sun Microsystems.

---

# Contents

Preface .....	xi
<b>Chapter 1 Introduction to UNIX Text Editing .....</b>	<b>3</b>
1.1. Available Editors .....	3
1.2. What to Do If Something Goes Wrong .....	4
1.3. Other Text-Handling Programs .....	5
<b>Chapter 2 Using vi, the Visual Display Editor .....</b>	<b>13</b>
2.1. vi and ex .....	13
2.2. Getting Started .....	14
Editing a File .....	14
The Editor's Copy — Editing in the Buffer .....	14
Arrow Keys .....	15
Special Characters: ESC, CR and CTRL-C .....	15
Getting Out of vi — :q, :q!, :w, ZZ, :x, :wq .....	15
2.3. Moving Around in the File .....	16
Scrolling and Paging — CTRL-D, CTRL-U, CTRL-E, CTRL-Y, CTRL-F, CTRL-B .....	16
Searching, Goto, and Previous Context — /, ?, G .....	16
Moving Around on the Screen — h, j, k, l, +, -, H, M, L .....	18
Moving Within a Line — b, w, e, E, B, W .....	18
Viewing a File — view .....	19
2.4. Making Simple Changes .....	19
Inserting — i, I, a, A, o, and O .....	19
Making Small Corrections — x, r, s, R .....	20

Deleting, Repeating, and Changing — <code>dw</code> , <code>.</code> , <code>db</code> , <code>c</code> .....	21
Operating on Lines — <code>dd</code> , <code>cc</code> , <code>S</code> .....	21
Undoing — <code>u</code> , <code>U</code> .....	21
2.5. Moving About: Rearranging and Duplicating Text .....	22
Low-level Character Motions — <code>f</code> , <code>F</code> , <code>^</code> .....	22
Higher Level Text Objects — <code>(</code> , <code>)</code> , <code>{</code> , <code>}</code> , <code>[[</code> , <code>]]</code> .....	23
Rearranging and Duplicating Text — <code>y</code> , <code>Y</code> , <code>p</code> , <code>P</code> .....	23
2.6. High-Level Commands .....	24
Writing, Quitting, and Editing New Files — <code>ZZ</code> , <code>:w</code> , <code>:q</code> , <code>:e</code> , <code>:n</code> .....	24
Escaping to a Shell — <code>:!</code> , <code>:sh</code> , <code>CTRL-Z</code> .....	25
Marking and Returning — <code>m</code> .....	25
Adjusting the Screen <code>CTRL-L</code> , <code>z</code> .....	25
2.7. Special Topics .....	26
Options, the Set Variable, and Editor Start-up Files .....	26
Recovering Lost Lines .....	27
Recovering Lost Files — the <code>r</code> Option .....	28
Continuous Text Input — <code>wrapmargin</code> .....	28
Features for Editing Programs .....	28
Filtering Portions of the Buffer .....	29
Commands for Editing LISP .....	29
Macros .....	30
Word Abbreviations — <code>:ab</code> , <code>:una</code> .....	31
2.8. Nitty-gritty Details .....	31
Line Representation in the Display .....	32
Command Counts .....	32
File Manipulation Commands .....	33
More about Searching for Strings .....	34
More about Input Mode .....	35
2.9. Command and Function Reference .....	37
Notation .....	37
Commands .....	38
Entry and Exit .....	38
Cursor and Page Motion .....	38

Searches .....	41
Text Insertion .....	42
Text Deletion .....	42
Text Replacement .....	42
Moving Text .....	43
Miscellaneous Commands .....	44
Special Insert Characters .....	45
: Commands .....	46
Set Commands .....	47
Character Functions .....	51
2.10. Terminal Information .....	61
Specifying Terminal Type .....	61
Special Arrangements for Startup .....	62
Open Mode on Hardcopy Terminals and 'Glass tty's' .....	62
Editing on Slow Terminals .....	63
Upper-case Only Terminals .....	64
2.11. Command Summary .....	64
<b>Chapter 3 Command Reference for the <code>ex</code> Line Editor .....</b>	<b>71</b>
3.1. Using <code>ex</code> .....	71
3.2. File Manipulation .....	72
Current File .....	72
Alternate File .....	72
Filename Expansion .....	72
3.3. Special Characters .....	73
Multiple Files and Named Buffers .....	73
Read-Only Mode .....	73
3.4. Exceptional Conditions .....	73
Errors and Interrupts .....	73
Recovering If Something Goes Wrong .....	73
3.5. Editing Modes .....	74
3.6. Command Structure .....	74
Specifying Command Parameters .....	74

Invoking Command Variants .....	75
Flags After Commands .....	75
Writing Comments .....	75
Putting Multiple Commands on a Line .....	75
Reporting Large Changes .....	75
3.7. Command Addressing .....	75
Addressing Primitives .....	75
Combining Addressing Primitives .....	76
3.8. Regular Expressions and Substitute Replacement Patterns .....	76
Regular Expressions .....	76
Magic and Nomagic .....	76
Basic Regular Expression Summary .....	77
Combining Regular Expression Primitives .....	77
Substitute Replacement Patterns .....	78
3.9. Command Reference .....	78
3.10. Option Descriptions .....	88
3.11. Limitations .....	94
<b>Chapter 4 Using the <code>ed</code> Line Editor .....</b>	<b>99</b>
4.1. Getting Started .....	99
Creating Text — the Append Command <code>a</code> .....	100
Error Messages — <code>?</code> .....	101
Writing Text Out as a File — the Write Command <code>w</code> .....	101
Leaving <code>ed</code> — the Quit Command <code>q</code> .....	102
Creating a New File — the Edit Command <code>e</code> .....	102
Exercise: Trying the <code>e</code> Command .....	103
Checking the Filename — the Filename Command <code>f</code> .....	104
Reading Text from a File — the Read Command <code>r</code> .....	104
Printing the Buffer Contents — the Print Command <code>p</code> .....	105
Exercise: Trying the <code>p</code> Command .....	106
Displaying Text — the List Command <code>l</code> .....	106
The Current Line — ‘Dot’ or ‘.’ .....	107
Deleting Lines — the Delete Command <code>d</code> .....	108

Exercise: Experimenting .....	109
Modifying Text — the Substitute Command <code>s</code> .....	109
The Ampersand <code>&amp;</code> .....	111
Exercise: Trying the <code>s</code> and <code>g</code> Commands .....	112
Undoing a Command — the Undo Command <code>u</code> .....	112
4.2. Changing and Inserting Text — the <code>c</code> and <code>i</code> Commands .....	113
Exercise: Trying the <code>c</code> Command .....	113
4.3. Specifying Lines in the Editor .....	114
Context Searching .....	114
Exercise: Trying Context Searching .....	115
Specifying Lines with Address Arithmetic — <code>+</code> and .....	115
Repeated Searches — <code>//</code> and <code>??</code> .....	117
Default Line Numbers and the Value of Dot .....	117
Combining Commands — the Semicolon <code>;</code> .....	119
Interrupting the Editor .....	120
4.4. Editing All Lines — the Global Commands <code>g</code> and <code>v</code> .....	121
Multi-line Global Commands .....	122
4.5. Special Characters .....	123
Matching Anything — the Dot <code>.</code> .....	123
Specifying Any Character — the Backslash <code>\</code> .....	124
Specifying the End of Line — the Dollar Sign <code>\$</code> .....	126
Specifying the Beginning of the Line — the Circumflex <code>^</code> .....	127
Matching Anything — the Star <code>*</code> .....	127
Character Classes — Brackets <code>[ &lt; ]</code> .....	129
4.6. Cutting and Pasting with the Editor .....	130
Moving Lines Around .....	130
Moving Text Around — the Move Command <code>m</code> .....	130
Substituting Newlines .....	132
Joining Lines — the Join Command <code>j</code> .....	132
Rearranging a Line with <code>&lt;\ ( ... \)</code> .....	133
Marking a Line — the Mark Command <code>k</code> .....	133
Copying Lines — the Transfer Command <code>t</code> .....	134
4.7. Escaping to the Shell with <code>!</code> .....	134

4.8. Supporting Tools .....	135
Editing Scripts .....	135
Matching Patterns with <code>grep</code> .....	135
4.9. Summary of Commands and Line Numbers .....	136
<b>Chapter 5 Using <code>view</code> .....</b>	<b>141</b>
<b>Chapter 6 Using <code>cat</code> .....</b>	<b>145</b>
<b>Chapter 7 Using <code>head</code> .....</b>	<b>149</b>
<b>Chapter 8 Using <code>tail</code> .....</b>	<b>153</b>
<b>Chapter 9 Using <code>more</code> .....</b>	<b>157</b>

---

## Tables

Table 2-1 Editor Options .....	26
Table 2-2 File Manipulation Commands .....	33
Table 2-3 Extended Pattern Matching Characters .....	35
Table 2-4 Input Mode Corrections .....	36
Table 2-5 Common Character Abbreviations .....	37
Table 2-6 Terminal Types .....	61
Table 2-7 Frequently-Used vi Commands .....	65



---

# Preface

This manual describes the features and options provided by the text editors available in the standard Sun system software. All you need to use one of these editors is:

- a Sun workstation (or terminal connected to a Sun workstation), and
- Sun system software.

## Purpose of This Manual

The purpose of this manual is to enable Sun workstation users to use the Sun system editor of their choice. Editing novices are referred to more elementary texts.

## Assumptions About the Reader

We assume you are familiar with a terminal keyboard and the basic structure of UNIX.† If you don't know how to log in or you don't understand how the file system is structured in UNIX, see *Getting Started with UNIX: Beginner's Guide*. If you have never used any of the UNIX text editors, start with the "Editing Files" chapter in *Getting Started with UNIX: Beginner's Guide* and also read the introductory chapter in this manual. Finally, we assume that you are using a Sun Workstation, although specific terminal information is also provided.

## Suggestions

The information in this manual is presented in two styles:

- examples and exercises, and
- reference.

As you are reading sections of this manual, sit down at your workstation and try the exercises and examples. The reference sections serve two functions: they explain in greater detail how to use the most-often-used features, and also cover the less-often-used features and options. Another reference source for additional details on Sun system commands and programs is the *Commands Reference Manual for the Sun Workstation*.

## Summary of Contents

This manual is divided into two major sections: Section I: Basic Editors and Section II: Viewing Files. Section I covers the standard text editors, presented in decreasing order of use. Section II describes various methods of viewing files or parts of files in read-only mode. Section I contains four chapters and two

---

† UNIX is a trademark of AT&T Bell Laboratories.

quick reference sheets:

Chapter 1 — *Introduction to UNIX Text Editing* — provides a guide to the available editing tools, and suggestions for what to do if something goes wrong in the editor. Newcomers to editing on a Sun workstation should start here.

Chapter 2 — *Using vi, the Visual Display Editor* — provides tutorial and reference information on the visual display editor `vi`.

Quick Reference — `vi`

Chapter 3 — *Command Reference for the ex Line Editor* — is a command reference for the `ex` and `vi` editors.

Quick Reference — `ex`

Chapter 4 — *Using the ed Line Editor* — provides a user's guide to the `ed` tools.

Section II contains information on `view`, `cat`, `head`, `tail`, and more.

#### Conventions Used in This Manual

Throughout this manual we use

`hostname%`

as the prompt to which you type system commands. **Bold face typewriter font** indicates commands that you type in exactly as printed on the page of this manual. Regular typewriter font represents what the system prints out to your screen. Typewriter font also specifies Sun system command names (program names) and illustrates source code listings. *Italics* indicates general arguments or parameters that you should replace with a specific word or string. We also occasionally use italics to emphasize important terms.

## Introduction to UNIX Text Editing

Introduction to UNIX Text Editing .....	3
1.1. Available Editors .....	3
1.2. What to Do If Something Goes Wrong .....	4
1.3. Other Text-Handling Programs .....	5



---

# Introduction to UNIX Text Editing

If you are familiar with the UNIX text editors, you can refer directly to the chapter in this manual covering the specific editor in which you are interested.

If you want a quick reference to remind you of a particular feature of either `vi` or `ex`, you can refer to one of the summary sheets for these editors, following the `vi` and `ex` chapters, respectively.

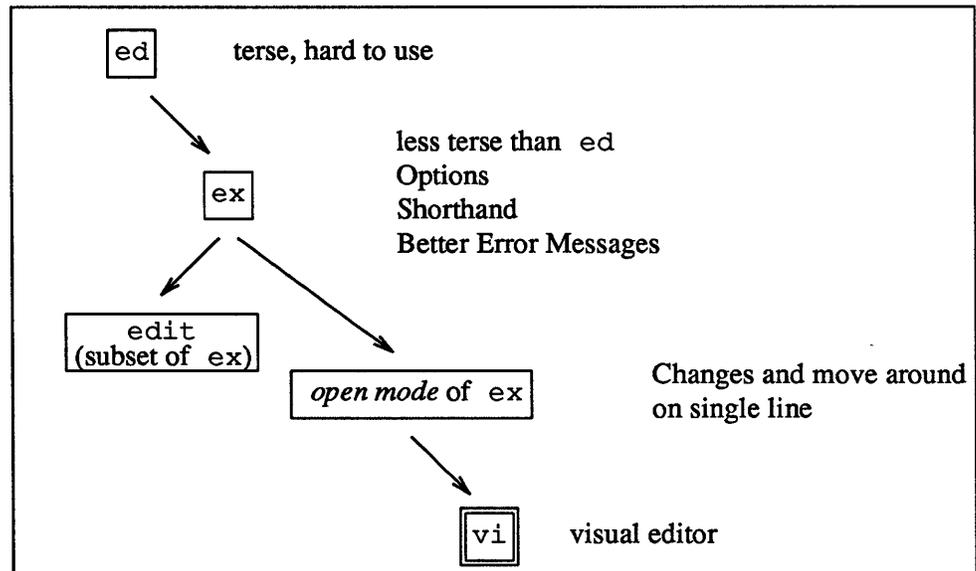
If you are *not* familiar with UNIX text editors, read the chapter "Editing Files" in *Getting Started with UNIX: Beginner's Guide*. It provides a good introduction to `vi`, including how to create a file, move the cursor around within a file, and change the contents of a file.

## 1.1. Available Editors

UNIX provides the text editors `vi`, `ex`, `edit`, and `ed`. The one you will probably use most often is `vi`, although you might want to choose to use the one that is most familiar to you.

`vi` is a screen-oriented, or display editor. It is "built on top of" `ex`, which means `vi` has almost all of the features of `ex`, plus many more features that are specific to `vi`. You can issue almost all `ex` commands from within `vi` by preceding them with the colon (`:`) character. Only a few `ex` commands require you to invoke the `ex` editor explicitly. `edit` is a subset of `ex`, and is therefore covered in that chapter.

The following diagram briefly sketches the history of UNIX text editors. `ed` was the original line editor, after which all the others were modeled. `ex` improved on `ed`, by being less terse, and providing display options like numbered lines, by allowing shorthand versions of commands, and by responding with clearer error messages. However, `ex` is still a line editor. `edit`, another line editor, provides a subset of `ex`'s features. Later, *open mode* was added to `ex`, which enabled the user to make changes and move the cursor around on only a single line at a time. The most advanced and most widely-used UNIX text editor as of this writing is the screen-oriented editor `vi`.



## 1.2. What to Do If Something Goes Wrong

In case you make a mistake, or something goes wrong, here are some suggestions for what to do.

If you make a mistake in the editor that you cannot recover from easily, don't panic. As long as you don't *write* the file *and quit* the editor, you can retrieve the original file. In such a case you have two choices:

- 1) Write ALL (good and bad) changes you just made to a new file and quit editing the current file, or
- 2) Quit the editor without saving any changes you just made.

You should write the entire edit buffer into a new file as long as most of the new changes are valid. In `vi`, the commands are `:w newfilename` to write the new file, then `:q!` to get out of the editor. You should quit without saving ANY of the changes if you know they are all wrong. In `vi`, the command is `:q!`.

Occasionally, you can get into a state where your workstation or terminal acts strangely. For example, you may not be able to move the cursor, or your cursor may disappear, or the terminal won't echo what you type, or typing RETURN may not cause a linefeed or return the cursor to the left margin. After a suitable length of time, try the following solutions:

- First, type CTRL-Q. In case you had accidentally typed CTRL-S, freezing the screen, this would resume the suspended output. Typing CTRL-Q would also resume suspended output from accidentally typing a NO SCRL key on your keyboard (also called SET UP/NO SCROLL on some terminals). This also freezes the keyboard like typing a CTRL-S.
- Next, try pressing the LINEFEED key, followed by typing RESET, and pressing LINEFEED again.
- If that doesn't help, try logging out and logging back in. If you are using a terminal, try powering it off and on to regain normal operation.

- If you get unwanted messages or garbage on your screen, type CTRL-L to refresh the workstation screen. (Use CTRL-R on a terminal.)

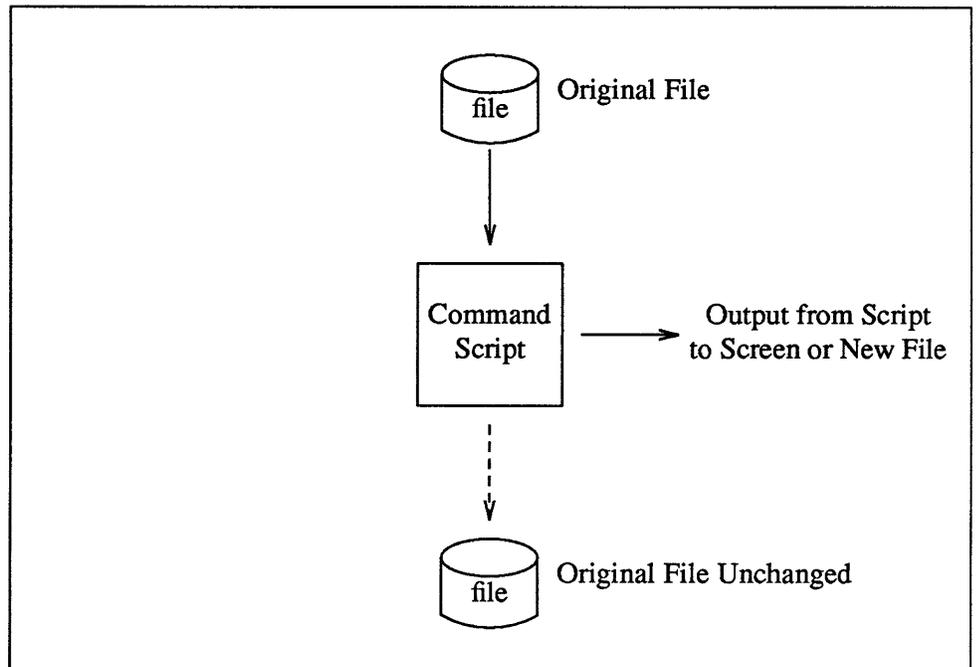
If your system goes down, the edit buffer is automatically saved in a file. Depending on the elapsed time since your last change, most to all of your latest changes are recoverable. After rebooting your system, or doing whatever needs to be done, you will receive mail indicating that the file has been saved. The mail message contains instructions on how to recover the file with your edits in it. First, return to the directory where the file belongs. Then, re-enter the editor with the `-r` option to *restore* the file:

```
hostname% vi -r filename
hostname%
```

When you are ready, write the changes to the file by typing `:w` or `:wq`.

### 1.3. Other Text-Handling Programs

Other UNIX utility programs such as `awk`, `sed`, `grep`, `fgrep`, `egrep`, and `tr` operate on a text file, but do not change the original file. You pass the file to be "edited" through a *script* (such as `awk` or `sed`) or *command* (such as `grep`) and the "changes" appear on your screen, but the file remains intact. Refer to the following diagram for an outline of how these utility programs work.



For more information, refer to *Using UNIX Text Utilities on the Sun Workstation*.



Section I:

Basic Editors



---

## Using `vi`, the Visual Display Editor

Using <code>vi</code> , the Visual Display Editor .....	13
2.1. <code>vi</code> and <code>ex</code> .....	13
2.2. Getting Started .....	14
Editing a File .....	14
The Editor's Copy — Editing in the Buffer .....	14
Arrow Keys .....	15
Special Characters: ESC, CR and CTRL-C .....	15
Getting Out of <code>vi</code> — <code>:q</code> , <code>:q!</code> , <code>:w</code> , <code>ZZ</code> , <code>:x</code> , <code>:wq</code> .....	15
2.3. Moving Around in the File .....	16
Scrolling and Paging — CTRL-D, CTRL-U, CTRL-E, CTRL-Y, CTRL-F, CTRL-B .....	16
Searching, Goto, and Previous Context — <code>/</code> , <code>?</code> , <code>G</code> .....	16
Moving Around on the Screen — <code>h</code> , <code>j</code> , <code>k</code> , <code>l</code> , <code>+</code> , <code>-</code> , <code>H</code> , <code>M</code> , ..... <code>L</code> .....	18
Moving Within a Line — <code>b</code> , <code>w</code> , <code>e</code> , <code>E</code> , <code>B</code> , <code>W</code> .....	18
Viewing a File — <code>view</code> .....	19
2.4. Making Simple Changes .....	19
Inserting — <code>i</code> , <code>I</code> , <code>a</code> , <code>A</code> , <code>o</code> , and <code>O</code> .....	19
Making Small Corrections — <code>x</code> , <code>r</code> , <code>s</code> , <code>R</code> .....	20
Deleting, Repeating, and Changing — <code>dw</code> , <code>.</code> , <code>db</code> , <code>c</code> .....	21
Operating on Lines — <code>dd</code> , <code>cc</code> , <code>S</code> .....	21
Undoing — <code>u</code> , <code>U</code> .....	21
2.5. Moving About: Rearranging and Duplicating Text .....	22
Low-level Character Motions — <code>f</code> , <code>F</code> , <code>^</code> .....	22

Higher Level Text Objects — (, ), {, }, [[, ]]	23
Rearranging and Duplicating Text — y, Y, p, P	23
2.6. High-Level Commands	24
Writing, Quitting, and Editing New Files — ZZ, :w, :q, :e, :n	24
Escaping to a Shell — :!, :sh, CTRL-Z	25
Marking and Returning — m	25
Adjusting the Screen CTRL-L, z	25
2.7. Special Topics	26
Options, the Set Variable, and Editor Start-up Files	26
Recovering Lost Lines	27
Recovering Lost Files — the r Option	28
Continuous Text Input — wrapmargin	28
Features for Editing Programs	28
Filtering Portions of the Buffer	29
Commands for Editing LISP	29
Macros	30
Word Abbreviations — :ab, :una	31
2.8. Nitty-gritty Details	31
Line Representation in the Display	32
Command Counts	32
File Manipulation Commands	33
More about Searching for Strings	34
More about Input Mode	35
2.9. Command and Function Reference	37
Notation	37
Commands	38
Entry and Exit	38
Cursor and Page Motion	38
Searches	41
Text Insertion	42
Text Deletion	42
Text Replacement	42
Moving Text	43
Miscellaneous Commands	44

Special Insert Characters .....	45
: Commands .....	46
Set Commands .....	47
Character Functions .....	51
2.10. Terminal Information .....	61
Specifying Terminal Type .....	61
Special Arrangements for Startup .....	62
Open Mode on Hardcopy Terminals and 'Glass tty's' .....	62
Editing on Slow Terminals .....	63
Upper-case Only Terminals .....	64
2.11. Command Summary .....	64



---

## Using `vi`, the Visual Display Editor

This chapter<sup>1</sup> describes `vi` (pronounced *vee-eye*) the visual, display editor. The first part of this chapter provides the basics of using `vi`. The second part provides a command reference and terminal set-up information. Finally, there is a quick reference, summarizing the `vi` commands. Keep this reference handy while you are learning `vi`. As the `vi` editor is the visual display version of the `ex` line editor, and because the full command set of the line-oriented `ex` editor is available within `vi`, you can use the `ex` commands in `vi`. Some editing, such as global substitution, is more easily done with `ex`. So refer to the information in the chapter "Command Reference for the `ex` Line Editor" as it also applies to `vi`.

This chapter assumes you are using `vi` on the Sun Workstation. If you are using `vi` on a terminal, refer to the "Terminal Information" section for instructions on setting up your terminal.

In the examples, input that must be typed as is will be presented in **bold typewriter font**. Text you should replace with appropriate input is given in *italics*.

### 2.1. `vi` and `ex`

As noted above, `vi` is actually one mode of editing within the editor `ex`. When you are running `vi` you can escape to the line-oriented editor `ex` by typing `Q`. All of the `:` commands introduced in the section on "File Manipulation Commands" are available in `ex`. This places the cursor on the command line at the bottom of the screen. Likewise, most `ex` commands can be invoked from `vi` using `:.`  Just give them without the `:` and follow them with a CR.

In rare instances, an internal error may occur in `vi`. In this case you will get a diagnostic and be left in the command mode of `ex`. You can then save your work and quit if you wish by giving the command `x` after the `:` that `ex` prompts you with, or you can re-enter `vi` by giving `ex` a `vi` command.

There are a number of things you can do more easily in `ex` than in `vi`. Systematic changes in line-oriented material are particularly easy. Experienced users often mix their use of `ex` command mode and `vi` command mode to speed the work they are doing. Keep these things in mind as you read on.

---

<sup>1</sup> The material in this chapter is derived from *An Introduction to Display Editing with Vi*, W.N. Joy, M. Horton, University of California, Berkeley and *Vi Command and Function Reference*, A.P.W. Hewett, M. Horton.

## 2.2. Getting Started

When using `vi`, changes you make to the file you are editing are reflected in what you see on your workstation screen.

During an editing session, there are two usual modes of operation: *command* mode and *insert* mode. In command mode you can move the cursor around in the file. There are commands to move the cursor forward and backward in units of characters, words, sentences and paragraphs. A small set of operators, like `d` for delete and `c` for change, are combined with the motion commands to form operations such as delete word or change paragraph. You can do other operations that do not involve entering fresh text. To enter new text into the file, you must be in insert mode. You get into insert mode with the `a` or `A` (append), `o` or `O` (open) and `I` or `i` (insert) commands. You get out of insert mode by typing the ESC (escape) key (or ALT on some keyboards). The significant characteristic of insert mode is that commands can't be used, so anything you type except ESC is inserted into the file. If you change your mind anytime in insert mode using `vi`, typing ESC cancels the command you started and reverses to command mode. If you had already typed some characters, typing `u` *undoes* the last insert operation. Also, if you are unsure of which mode you are in, type ESC until the screen flashes; this means that you are back in command mode.

Run `vi` on a copy of a file you are familiar with while you are reading this. Try the commands as they are described.

## Editing a File

To use `vi` on the file, type:

```
hostname% vi filename
```

replacing *filename* with the name of the file copy you just created. The screen clears and the text of your file appears.

If you do not get the display of text, you may have typed the wrong filename. `vi` has created a new file for you with the indication `''file'' [New file]`. Type `:q` (colon and the 'q' key) and then type the RETURN key. This should get you back to the command level interpreter. Then try again, this time spelling the filename correctly.

If `vi` doesn't seem to respond to the commands you type here, try sending `vi` an interrupt by typing a CTRL-C (or INTERRUPT signal) at your workstation (or by pressing the DEL or RUB keys on your terminal). Then type the `:q` command again followed by a RETURN. If you are using a terminal and something else happens, you may have given the system an incorrect terminal type code. `vi` may make a mess out of your screen. This happens when it sends control codes for one kind of terminal to some other kind of terminal. Type a `:q` and RETURN. Figure out what you did wrong (ask someone else if necessary) and try again.

## The Editor's Copy — Editing in the Buffer

`vi` does not directly modify the file you are editing. Rather, `vi` makes a copy of this file in a place called the *buffer*, and remembers the file's name. All changes you make while editing only change the contents of the buffer. You do not affect the contents of the file unless and until you *write* the buffer back into the original file.

## Arrow Keys

The editor command set is independent of the workstation or terminal you are using. On most terminals with cursor positioning keys, these keys will also work within the editor.<sup>2</sup> If you don't have cursor positioning keys, that is, keys with arrows on them, or even if you do, you can use the `h`, `j`, `k`, and `l` keys as cursor positioning keys. As you will see later, `h` moves back to the left (like CTRL-H, a backspace), `j` moves down (in the same column), `k` moves up (in the same column), and `l` moves the cursor to the right.

## Special Characters: ESC, CR and CTRL-C

Several of these special characters are very important, so be sure to find them right now. Look on your keyboard for a key labelled ESC (or ALT on some terminals). It is near the upper left corner of your workstation keyboard. Try typing this key a few times. `vi` flashes the screen (or beeps) to indicate that it is in a quiescent state. You can cancel partially formed commands with ESC. When you insert text in the file, you end the text insertion with ESC. This key is a fairly harmless one to press, so you can just press it until the screen flashes if you don't know what is going on.

Use RETURN (or CR for *carriage return*) key to terminate certain commands. It is at the right side of the workstation keyboard, and is the same key used at the end of each shell command.

Use the special character CTRL-C (or DEL or RUB key), to send an interrupt, to tell `vi` to stop what it is doing. It is a forceful way of making `vi` listen to you, or to return `vi` to the quiescent state if you don't know or don't like what is going on.

Try typing the `/` key on your keyboard. Use this key to search for a string of characters. `vi` displays the cursor at the bottom line of the screen after a `/` is displayed as a prompt. You can get the cursor back to the current position by pressing BACK SPACE (or DEL); try this now. This cancels the search. Typing CTRL-C also cancels the search. From now on we will simply refer to typing CTRL-C (or pressing the DEL or RUB key) as 'sending an interrupt.'<sup>3</sup>

`vi` often echoes your commands on the last line of the screen. If the cursor is on the first position of this last line, then `vi` is performing a computation, such as locating a new position in the file after a search or running a command to reformat part of the buffer. When this is happening, you can stop `vi` by sending an interrupt.

## Getting Out of vi — `:q`, `:q!`, `:w`, `ZZ`, `:x`, `:wq`

When you want to get out of `vi` and end the editing session, type `:q` to *quit*. If you have changed the buffer contents and type `:q`, `vi` responds with `No write since last change (:quit! overrides)`. If you then want to quit `vi` without saving the changes, type `:q!`. You need to know about `:q!` in case you change the editor's copy of a file you wish only to look at. Be very careful not to give this command when you really want to save the changes you have made.

<sup>2</sup> Note for the HP2621: on this terminal the function keys must be *shifted* to send to the machine, otherwise they only act locally. Unshifted use leaves the cursor positioned incorrectly.

<sup>3</sup> On some systems, this interruptibility comes at a price: you cannot type ahead when the editor is computing with the cursor on the bottom line.

Do not type :q! if you *want* to save your changes. To save or *write* your changes without quitting *vi*, type :w. While in the middle of an editing session, if you are sure about the changes you have made, it's a good idea to save your changes from time to time by typing :w.

To write the contents of the buffer back into the file you are editing, saving any changes you have made, and then to quit, type ZZ or :x. And finally, to write the file even if no changes have been made, and exit *vi*, type :wq.

You can terminate all commands that read from the last display line with an ESC as well as a RETURN.

### 2.3. Moving Around in the File

*vi* has a number of commands for moving around in the file. You can *scroll* forward and backward through a file, moving part of the text on the screen. You can *page* forward and backward through a file, by moving a whole screenful of text. You can also display one more line at the top or bottom of the screen.

#### Scrolling and Paging — CTRL-D, CTRL-U, CTRL-E, CTRL-Y, CTRL-F, CTRL-B

The most useful way to move through a file is to type the control (CTRL) and D keys at the same time, sending a CTRL-D. We use this notation to refer to control sequences from now on. When coupled with the CTRL key, the shift key is ignored, so CTRL-D and CTRL-d are equivalent.

Try typing CTRL-D to see that this command scrolls *down* in the file. The command to scroll *up* is CTRL-U. (Many dumb terminals cannot scroll up at all. In that case type CTRL-U to clear and refresh the screen, placing a line that is farther back in the file at the top of the screen.)

If you want to see more of the file below where you are, you can type CTRL-E to expose one more line at the bottom of the screen, leaving the cursor where it is. The CTRL-Y (which is hopelessly non-mnemonic, but next to CTRL-U on the keyboard) exposes one more line at the top of the screen.

You can also use the keys CTRL-F and CTRL-B to move *forward* and *backward* a page, keeping a couple of lines of continuity between screens so that it is possible to read through a file using these rather than CTRL-D and CTRL-U if you wish. CTRL-F and CTRL-B also take preceding counts, which specify the number of pages to move. For example, 2CTRL-F pages forward two pages.

Notice the difference between scrolling and paging. If you are trying to read the text in a file, typing CTRL-F to page forward leaves you only a little context to look back at. Scrolling with CTRL-D on the other hand, leaves more context, and moves more smoothly. You can continue to read the text as scrolling is taking place.

#### Searching, Goto, and Previous Context — /, ?, G

Another way to position yourself in the file is to give *vi* a string to search for. Type the character '/' followed by a string of characters terminated by RETURN. *vi* positions the cursor at the next occurrence of this string. Try typing n to go to the *next* occurrence of this string. The character '?' searches backward from where you are, and is otherwise like '/'. N is like n, but reverses the direction of the search.

You can string several search expressions together, separated by a semicolon in visual mode, the same as in command mode in `ex`. For example:

```
/today;/tomorrow
```

moves the cursor to the first 'tomorrow' after the next 'today'. This also works within one line.

These searches normally wrap around the end of the file, so you can find the string even if it is not on a line in the direction you search, provided it is somewhere else in the file. You can disable this *wraparound* with the command `:se nowrapscanCR`, or more briefly `:se nowsCR`.

If the search string you give `vi` is not present in the file, `vi` displays `Pattern not found` on the last line of the screen, and the cursor is returned to its initial position.

If you wish the search to match only at the beginning of a line, begin the search string with a caret character (^). To match only at the end of a line, end the search string with a dollar sign (\$). So to search for the word 'search' at the beginning of a line, type:

```
/^search<CR>
```

and to search for the word 'last' at the end of a line, type:

```
/last$<CR>
```

Actually, the string you give to search for here can be a *regular expression* in the sense of the editors `ex` and `ed`. If you don't wish to learn about this yet, you can disable this more general facility by typing

```
:se nomagic<CR>
```

By putting this command in `EXINIT` in your environment, you can have always this *nomagic* option in effect. See the section on "Special Topics" for details on how to do this.

The command `G`, when preceded by a number, positions the cursor at that line in the file. Thus `1G` moves the cursor to the first line of the file. If you do not give `G` any count, it positions you at the last line of the file.

If you are near the end of the file, and the last line is not at the bottom of the screen, `vi` places only the character tilde (~) on each remaining line. This indicates that the last line in the file is on the screen; that is, the ~ lines are past the end of the file.

You can find out the state of the file you are editing by typing a `CTRL-G`. `vi` shows you the name of the file you are editing, the number of the current line, the number of lines in the buffer, and the percentage of characters already displayed from the buffer. For example:

```
`data.file'' [Modified] line 329 of 1276 —8%—
```

Try doing this now, and remember the number of the line you are on. Give a `G` command to get to the end and then another `G` command with the line number to get back where you were.

You can get back to a previous position by using the command `' '` (two apostrophes). This returns you to the first non-blank space in the previous location. You can also use `` `` (two back quotes) to return to the previous position. The former is more easily typed on the keyboard. This is often more convenient than `G` because it requires no advance preparation. Try typing a `G` or a search with `/` or `?` and then a `` `` to get back to where you were. If you accidentally type `n` or any command that moves you far away from a context of interest, you can quickly get back by typing `` ``.

### Moving Around on the Screen

— `h`, `j`, `k`, `l`, `+`, `-`, `H`, `M`,  
`L`

Now try just moving the cursor around on the screen. Try the arrow keys as well as `h`, `j`, `k`, and `l`. You will probably prefer these keys to arrow keys, because they are right underneath your fingers. These are very common keys for moving up and down lines in the file. Notice that if you go off the bottom or top with these keys then the screen scrolls down (and up if possible) to bring a line at a time into view.

Type the `+` key. Each time you do, notice that the cursor advances to the next line in the file, at the first non-blank position on the line. The `-` key is like `+` but the cursor goes to the first non-blank character in the line above.

The RETURN key has the same effect as the `+` key.

`vi` also has commands to take you to the top, middle and bottom of the screen. `H` takes you to the top (*home*) line on the screen. Try preceding it with a number as in `3H`. This takes you to the third line on the screen. Try `M`, which takes you to the middle line on the screen, and `L`, which takes you to the *last* line on the screen. `L` also takes counts, so `5L` takes you to the fifth line from the bottom.

### Moving Within a Line — `b`,

`w`, `e`, `E`, `B`, `W`

Now pick a word on some line on the screen, not the first word on the line. Move the cursor using `h`, `j`, `k`, `l` or RETURN and `-` to be on the line where the word is. Try typing the `w` key. This advances the cursor to the next *word* on the line. `W` advances to the next word ignoring any punctuation. Try typing the `b` key to *back* up words in the line. Also try the `e` key which advances you to the *end* of the current word rather than to the beginning of the next word. Also try SPACE (the space bar) which moves right one character and the BACKSPACE (or CTRL-H) key which moves left one character. The key `h` works as CTRL-H does and is useful if you don't have a BACKSPACE key.

If the line had punctuation in it, you may have noticed that the `w` and `b` keys stopped at each group of punctuation. You can also go backward and forward words without stopping at punctuation by using `W` and `B` rather than the lower case equivalents. You can think of these as bigger words. The `E` command advances to the end of the current word, but unlike `e`, ignores punctuation. Try these on a few lines with punctuation to see how they differ from the lower case `e`, `w`, and `b`.

The word keys wrap around the end of line, rather than stopping at the end. Try moving to a word on a line below where you are by repeatedly typing `w`.

**Viewing a File — view**

If you want to use the editor to look at a file, rather than to make changes, use `view` instead of `vi`. This sets the *readonly* option which prevents you from accidentally overwriting the file. For example, to look at a file called *kubla*, type:

```
hostname% view kubla
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
``kubla`` [Read only] 5 lines, 149 characters
hostname%
```

To scroll through a file longer than one screenful, use the characters described in the previous section on "Scrolling and Paging". To get out of `view`, type `:q`. If you accidentally made changes to the file while the *readonly* option was set, type `:q!` to exit.

**2.4. Making Simple Changes**

Simple changes involve inserting, deleting, repeating, and changing single characters, words, and lines of text. In `vi`, you can also undo the previous change with ease in case you change your mind.

**Inserting — i, I, a, A, o, and O**

There are two basic commands for inserting new text: `i` to *insert* text to the left of the cursor, and `a` to *append* text to the right of the cursor. After you type `i`, everything you type until you press ESC is inserted into the file. Try this now; position yourself at some word in the file and try inserting text before this word. (If you are on a dumb terminal it will seem, for a minute, that some of the characters in your line have been overwritten, but they will reappear when you type ESC.)

Now try finding a word that can, but does not, end in an 's'. Position the cursor at this word and type `e` (move to end of word), then `a` (for append), 's', and ESC to terminate the text insert. Use this sequence of commands to easily make a word plural.

Try inserting and appending a few times to make sure you understand how this works.

It is often the case that you want to add new lines to the file you are editing, before or after some specific line in the file. Find a line where this makes sense and then give the command `o` to create a new line *after* the line you are on, or the command `O` to create a new line *before* the line you are on. After you create a new line in this way, text you type up to an ESC is inserted on the new line.

Many related editor commands are invoked by the same letter key and differ only in that one is given by a lower-case key and the other is given by an upper-case key. In these cases, the upper-case key often differs from the lower-case key in its sense of direction, with the upper-case key working backward or up, while the lower-case key moves forward or down.

Whenever you are typing in text, you can give many lines of input or just a few characters. To type in more than one line of text, type a RETURN at the middle of your input. A new line will be created for text, and you can continue to type. (If

you are on a slow, dumb terminal `vi` may choose to wait to redraw the tail of the screen, and will let you type over the existing screen lines. This avoids the lengthy delay that would occur if `vi` attempted to always keep the tail of the screen up to date. The tail of the screen will be fixed up, and the missing lines will reappear, when you type ESC.)

While you are inserting new text, you can use the DEL key at the system command level to backspace over the last character you typed. (This may be CTRL-H on a terminal.) Use CTRL-U (this may be CTRL-X on a terminal) to erase the input you have typed on the current line. In fact, the character CTRL-H (backspace) always works to erase the last input character here, regardless of what your erase character is.

CTRL-W erases a whole word and leaves you after the space after the previous word; use it to quickly back up when inserting.

Notice that when you backspace during an insertion, the characters you backspace over are not erased; the cursor moves backward, and the characters remain on the display. This is often useful if you are planning to type in something similar. In any case the characters disappear when when you press ESC; if you want to get rid of them immediately, hit an ESC and then `a` again.

Notice also that you can't erase characters you didn't insert, and that you can't backspace around the end of a line. If you need to back up to the previous line to make a correction, just hit ESC and move the cursor back to the previous line. After making the correction you can return to where you were and use the insert or append command again.

### Making Small Corrections —

`x`, `r`, `s`, `R`

You can make small corrections in existing text quite easily. Find a single character that is wrong or just pick any character. Use the arrow keys to find the character, or get near the character with the word motion keys and then either backspace with `h` (or the BACKSPACE key or CTRL-H) or type a SPACE (using the space bar) until the cursor is on the character that is wrong. If the character is not needed, type the `x` key; this deletes the character from the file. It is analogous to the way you `x` out characters when you make mistakes on a typewriter, except it's not as messy.

If a single character is incorrect, you can *replace* it with the correct character by typing the command `rc`, where `c` is replaced by the correct character. You don't need to type ESC. If you want to replace or type over more than one character, type `R` and then the ESC key to get out of insert mode when you are finished. Finally if the character that is incorrect should be replaced by more than one character, type `s` which *substitutes* for the single character, a string of characters, and end the substitution with ESC. If there are a small number of characters that are wrong you can precede `s` with a count of the number of characters to be replaced. You can use counts with `x` to specify the number of characters to be deleted and with `r`, such as `4rx` to specify that a character be replaced with four `x`'s.

Use `xp` to correct simple typos in which you have inverted the order of two letters. The `p` for *put* is described later.

### Deleting, Repeating, and Changing — `dw`, `.`, `db`, `c`

You already know almost enough to make changes at a higher level. All you need to know now is that the `d` key acts as a delete operator. Try the command `dw` to *delete a word*. Try typing `.` a few times. Notice that this repeats the effect of the `dw`. The `.` repeats the last command that made a change. You can remember it by analogy with an ellipsis `'...'`.

Now try `db`. This deletes a word before the cursor, namely the preceding word. Try `dSPACE`. This deletes a single character, and is equivalent to the `x` command.

Use `D` to delete the rest of the line the cursor is on.

Another very useful operator is `c` or *change*. Thus `cw` changes the text of a single word. You follow it by the replacement text ending with an ESC. Find a word that you can change to another, and try this now. Notice that the end of the text to be changed is marked with the dollar sign character (`$`) so that you can see this as you are typing in the new material.

### Operating on Lines — `dd`, `cc`, `S`

It is often the case that you want to operate on lines. Find a line you want to delete, and type `dd`, the `d` operator twice. This deletes the line.

If you are on a dumb terminal, `vi` may just erase the line on the screen, replacing it with a line with only an at-sign (`@`) on it. This line does not correspond to any line in your file, but only acts as a place holder. It helps to avoid a lengthy redraw of the rest of the screen which would be necessary to close up the hole created by the deletion on a terminal without a delete line capability.

Try repeating the `c` operator twice; this changes a whole line, erasing its previous contents and replacing them with text you type up to an ESC. The command `S` is a convenient synonym for `cc`, by analogy with `s`. Think of `S` as a substitute on lines, while `s` is a substitute on characters.

You can delete or change more than one line by preceding the `dd` or `cc` with a count, such as `5dd`, which deletes 5 lines. You can also give a command like `dL` to delete all the lines up to and including the *last* line on the screen, or `d3L` to delete through the third from the bottom line. Try some commands like this now.<sup>4</sup> Notice that `vi` lets you know when you change a large number of lines so that you can see the extent of the change. It also always tells you when a change you make affects text you cannot see.

### Undoing — `u`, `U`

Now suppose that the last change you made was incorrect; you could use the insert, delete and append commands to put the correct material back. However, since it is often the case that we regret a change or make a change incorrectly, `vi` provides a `u` command to *undo* the last change you made. Try this a few times, and give it twice in a row to notice that a `u` also undoes a `u`.

The undo command lets you reverse only a single change. After you make a number of changes to a line, you may decide that you would rather have the

<sup>4</sup> One subtle point here involves using the `/'` search after a `d`. This normally deletes characters from the current position to the point of the match. If what is desired is to delete whole lines including the two points, give the pattern as `/pat/+0`, a line address.

original state of the line back. The `U` command restores the current line to the state before you started changing it, only as long as you do not move the cursor off the line. If you move the cursor away from the line you changed, `U` does nothing.

You can recover text that you delete, even if `u` (undo) will not bring it back; see the section on "Recovering Lost Lines" on how to recover lost text.

## 2.5. Moving About: Rearranging and Duplicating Text

### Low-level Character Motions

— `f`, `F`, `^`

This describes more commands for moving in a file and explains how to rearrange and make copies of text.

Now move the cursor to a line where there is a punctuation or a bracketing character such as a parenthesis, a comma or a period. Try the command `f x` to *find* the next *x* character to the right of the cursor in the current line. Try then hitting a `;` which finds the next instance on that line of the same character. By using the `f` command and then a sequence of `;`s you can often get to a particular place in a line much faster than with a sequence of word motions or SPACES. There is also an `F` command, which is like `f`, but searches backward. After instituting a search, the `;` repeats the search in the same direction as it was begun, and a comma (`,`) repeats the search in the opposite direction.

When you are operating on the text in a line, it is often desirable to deal with the characters up to, but not including, the first instance of a character. Try `d f x` for some *x* now and notice that the *x* character is deleted. Undo this with `u` and then try `d t x`; the `t` here stands for *to*, that is, delete up to the next *x*, but *not* the *x*. The command `T` is the reverse of `t`.

When working with the text of a single line, a `^` moves the cursor to the first non-blank position on the line, and a `$` moves it to the end of the line. Thus `$a` appends new text at the end of the current line (as does `A` which is easier to type).

Your file may have tab (CTRL-I) characters in it. These characters are represented as a number of spaces expanding to a tab stop, where tab stops are every eight positions.<sup>5</sup> When the cursor is at a tab, it sits on the last of the several spaces that represent that tab. Try moving the cursor back and forth over tabs so you understand how this works.

On rare occasions, your file may have non-printing characters in it. These characters are displayed as control sequences, and look like a caret character (`^`) adjacent to another character. For example, the symbol for a new page (CTRL-L), looks like `^L` in the input file. However, spacing or backspacing over the character reveals that the two characters displayed represent only a single character.

The editor sometimes discards control characters, depending on the character and the setting of the *beautify* option, if you attempt to insert them in your file. You can get a control character in the file by beginning an insert and then typing a CTRL-V before the control character. The CTRL-V quotes the following character,

---

<sup>5</sup> You can set this with a command of the form `:set ts=x<CR>`, where *x* is four to set tabstops every four columns, for example. This affects the screen representation within the editor.

causing it to be inserted directly into the file.

### Higher Level Text Objects — (, ), {, }, [[, ]]

In working with a document it is often advantageous to work in terms of sentences, paragraphs, and sections. The operations ‘(’ and ‘)’ move to the beginning of the previous and next sentences respectively. Thus the command `d)` deletes the rest of the current sentence; likewise `d(` deletes the previous sentence if you are at the beginning of the current sentence, or the current sentence up to where you are if you are not at the beginning of the current sentence.

A sentence is defined as ending at a ‘.’, ‘!’ or ‘?’ followed by either the end of a line, or by two spaces. Any number of closing ‘)’, ‘]’, ‘”’ and ‘”’ characters may appear after the ‘.’, ‘!’ or ‘?’ before the spaces or end of the line.

The operations ‘{’ and ‘}’ move over paragraphs and the operations ‘[[’ and ‘]]’ move over sections. The ‘[[’ and ‘]]’ operations require the operation character to be doubled because they can move the cursor far from where it currently is. While it is easy to get back with the command ‘`’`, these commands would still be frustrating if they were easy to type accidentally.

A paragraph begins after each empty line, and also at each of a set of paragraph macros, specified by the pairs of characters in the definition of the string-valued option *paragraphs*. The default setting for this option defines the paragraph macros of the `-ms` macro package, that is the `.IP`, `.LP`, `.PP`, and `.QP` macros. You can easily change or extend this set of macros by assigning a different string to the *paragraphs* option in your EXINIT. See the section on "Special Topics" for details. The `.bp` directive is also considered to start a paragraph. Each paragraph boundary is also a sentence boundary. The sentence and paragraph commands take counts to operate over groups of sentences and paragraphs.

Sections in the editor begin after each macro in the *sections* option, normally `.NH` and `.SH`, and each line with a formfeed CTRL-L in the first column. Section boundaries are always line and paragraph boundaries also.

Try experimenting with the sentence and paragraph commands until you are sure how they work. If you have a large document, try looking through it using the section commands. The section commands interpret a preceding count as a different view size in which to redraw the screen at the new location, and this size is the base size for newly-drawn screens until another size is specified. (This is very useful if you are on a slow terminal and are looking for a particular section. You can give the first section command a small count to then see each successive section heading in a small screen area.)

### Rearranging and Duplicating Text — y, Y, p, P

`vi` has a single unnamed buffer where the last deleted or changed text is saved away, and a set of named buffers `a-z` that you can use to save copies of text and to move text around in your file and between files.

The operator `y` *yanks* a copy of the object that follows into the unnamed buffer. If preceded by a buffer name, `"xy`, where `x` here is replaced by a letter `a-z`, it places the text in the named buffer. The text can then be put back in the file with the commands `p` and `P`; `p` puts the text after or below the cursor, while `P` puts the text before or above the cursor.

If the text you yank forms a part of a line, or is an object such as a sentence that partially spans more than one line, then when you put the text back, it will be placed after the cursor (or before if you use `P`). If the yanked text forms whole lines, they will be put back as whole lines, without changing the current line. In this case, the put acts much like an `o` or `O` command.

Try the command `YP`. This makes a copy of the current line and leaves the cursor on this copy, which is placed before the current line. The command `Y` is a convenient abbreviation for `yy`. The command `Yp` will also make a copy of the current line, and place it after the current line. You can give `Y` a count of lines to yank, and thus duplicate several lines; try `3YP`.

To move text within the buffer, you need to delete it in one place, and put it back in another. You can precede a delete operation by the name of a buffer in which the text is to be stored as in `"a5dd` deleting 5 lines into the named buffer `a`. You can then move the cursor to the eventual resting place of the lines and do a `"ap` or `"aP` to put them back. In fact, you can switch and edit another file before you put the lines back, by giving a command of the form `:e nameCR` where `name` is the name of the other file you want to edit. You will have to write back the contents of the current editor buffer (or discard them) if you have made changes before `vi` will let you switch to the other file. An ordinary delete command saves the text in the unnamed buffer, so that an ordinary put can move it elsewhere. However, the unnamed buffer is lost when you change files, so to move text from one file to another you must use a named buffer.

## 2.6. High-Level Commands

Writing, Quitting, and Editing  
New Files — `ZZ`, `:w`, `:q`,  
`:e`, `:n`

A description of high-level commands that do more than juggle text follows.

So far you have seen how to enter `vi` and to write out your file using either `ZZ` or `:wCR`. The first exits from `vi`, writing if changes were made, and the second writes and stays in `vi`. We have also described that if you have changed the editor's copy of the file but do not wish to save your changes, either because you messed up the file or decided that the changes are not an improvement to the file, you type

```
:q!<CR>
```

to *quit* from the editor without writing the changes.

You can also re-edit the same file and start over by typing `:e!CR`. Use the `!` command rarely and with caution, as it is not possible to recover the changes you have made after you discard them in this manner.

You can also edit a different file without leaving `vi` by giving the command `:e nameCR`. If you have not written out your file before you try to do this, `vi` tells you this, ('No write since last change: (:edit! overrides)') and delays editing the other file. You can then type `:wCR` to save your work, followed by the `:e nameCR` command again, or carefully give the command `:e! nameCR`, which edits the other file discarding the changes you have made to the current file. To save changes automatically, include `set autowrite` in your EXINIT, and use `:n` instead of `:e`. See the "Special Topics" section for details on EXINIT.

### Escaping to a Shell — `!`, `:sh`, CTRL-Z

You can get to a shell to execute a single command by giving a `vi` command of the form `!cmdCR`. The system runs the single command `cmd` and when the command finishes, `vi` asks you to Press RETURN to continue. When you have finished looking at the output on the screen, type RETURN, and `vi` redraws the screen. You can then continue editing. You can also give another `:` command when it asks you for a RETURN; in this case the screen will not be redrawn.

If you wish to execute more than one command in the shell, give the command `:shCR`. This gives you a new shell, and when you finish with the shell, ending it by typing a CTRL-D, `vi` clears the screen and continues.

Use CTRL-Z to suspend `vi` and to return to the top level shell. The screen is redrawn when `vi` is resumed. This is the same as `:stop`.

### Marking and Returning — `m`

The command ```` returned to the previous place after a motion of the cursor by a command such as `/`, `?` or `G`. You can also *mark* lines in the file with single letter tags and return to these marks later by naming the tags. Try marking the current line with the command `mx`, where you should pick some letter for `x`, say `a`. Then move the cursor to a different line (any way you like) and type ``a`. The cursor will return to the place you marked. Marks last only until you edit another file.

When using operators such as `d` and referring to marked lines, it is often desirable to delete whole lines rather than deleting to the exact position in the line marked by `m`. In this case you can use the form ``x` rather than ``x`. Used without an operator, ``x` will move to the first non-blank character of the marked line; similarly ```` moves to the first non-blank character of the line containing the previous context mark ````.

### Adjusting the Screen CTRL-L, Z

If the screen image is messed up because of a transmission error to your workstation, or because some program other than `vi` wrote output to your workstation, you can type a CTRL-L, the ASCII form-feed character, to refresh the screen. (On a dumb terminal, if there are @ lines in the middle of the screen as a result of line deletion, you may get rid of these lines by typing CTRL-R to *retype* the screen, closing up these holes.<sup>6</sup>)

If you wish to place a certain line on the screen at the top middle or bottom of the screen, position the cursor to that line, and give a `z` command. Follow the `z` command with a RETURN if you want the line to appear at the top of the window, a `.` if you want it at the center, or a `-` if you want it at the bottom.

If you want to change the window size, use the `z` command as in `z5<CR>` to change the window to five lines.

<sup>6</sup> This includes Televideo 912/920 and ADM31 terminals.

## 2.7. Special Topics

There are several facilities that you can use to customize an editing session.

### Options, the Set Variable, and Editor Start-up Files

`vi` has a set of options, some of which have been mentioned above. The most useful options are described in the following table.

Table 2-1 *Editor Options*

Option	Default	Description
autoindent	noai	Supply indentation automatically
autowrite	noaw	Automatic write before :n, :ta, CTRL-^, !
ignorecase	noic	Ignore letter case in searching
lisp	nolisp	( { ) } commands deal with S-expressions
list	nolist	Tabs print as ^I, end of lines marked with \$
magic	magic	The characters . [ and * are special in scans
number	nonu	Lines are displayed prefixed with line numbers
paragraphs	para=IPLPPPQPbpP LI	Macro names that start paragraphs
redraw	nore	Simulate a smart terminal on a dumb one
sections	sect=NHSHH HU	Macro names that start new sections
shiftwidth	sw=8	Shift distance for <, > and input CTRL-D and CTRL-T
showmatch	nosm	Show matching ( or { as ) or } is displayed
slowopen	slow	Postpone display updates during inserts
term	dumb	The kind of terminal you are using.

The options are of three kinds: numeric options, string options, and toggle options. You can set numeric and string options by a statement of the form:

```
set opt=val
```

and toggle options can be set or unset by statements of one of the forms

```
set opt
set noopt
```

Put these statements in your EXINIT in your environment (described below), or use them while you are running `vi` by preceding them with a `:` and following them with a RETURN. For example, to display line numbers at the beginning of each line, use:

```
:se nu
```

You can get a list of all options that you have changed:

```
:set<CR>
redraw term=sun wrapmargin=8
```

or the value of a single option with

```
:set opt?<CR>
```

```
:set noai?<CR>
noautoindent
```

The

```
:set all<CR>
```

command generates a list of all possible options and their values. You can abbreviate `set` to `se`. You can also put multiple options on one line, such as,

```
:se ai aw nu<CR>
```

When you set options with the `set` command, they only last until you terminate the editing session in `vi`. It is common to want to have certain options set whenever you use the editor. To do this, create a list of `ex` commands to be run every time you start up `vi`, `ex`, or `edit`. All commands that start with a colon (`:`) are `ex` commands. A typical list includes a `set` command, and possibly a few `map` commands. Put these commands on one line by separating them with the pipe (`|`) character. If you use the `c` shell, `ssh`, put a line like this in the

```
setenv EXINIT 'set ai aw terse|map @ dd|map # x'
```

*This sets the options `autoindent`, `autowrite`, `terse`, (the `set` command), and makes `@` delete a line, (the first `map`), and makes `#` delete a character, (the second `map`). (See the "Macros" section for a description of the `map` command.)*

If you use the Bourne shell, put these lines in the file `.profile` in your home directory:

```
EXINIT='set ai aw terse|map @ dd|map # x'
export EXINIT
```

Of course, the particulars of the line depend on the options you want to set.

## Recovering Lost Lines

You might have a serious problem if you delete a number of lines and then regret that they were deleted. Despair not, `vi` saves the last nine deleted blocks of text in a set of numbered registers 1–9. You can get the *n*th previous deleted text back in your file by `"n p`. The `"` here says that a buffer name is to follow, *n* is the number of the buffer you wish to try (use the number 1 for now), and `p`, that *puts* text in the buffer after the cursor. If this doesn't bring back the text you wanted, type `u` to undo this and then (period) `.` to repeat the `p`. In general the `'.'` command repeats the last change you made. As a special case, when the last command refers to a numbered text buffer, the `'.'` command increments the number of the buffer before repeating the command. Thus a sequence of the form:

```
"1pu.u.u.
```

will, if repeated long enough, show you all the deleted text that has been saved for you. You can omit the `u` commands here to gather up all this text in the buffer, or stop after any `.` command to keep just the recovered text. You can also use `P` rather than `p` to put the recovered text before rather than after the cursor.

## Recovering Lost Files — the `-r` Option

If something goes wrong so the system goes down, you can recover the work you were doing up to the last few changes. You will normally receive mail when you next log in giving you the name of the file that has been saved for you. You should then change to the directory where you were when the system went down and type:

```
hostname% vi -r filename
```

replacing *filename* with the name of the file you were editing. This will recover your work to a point near where you left off. In rare cases, some of the lines of the file may be lost. `vi` will give you the numbers of these lines and the text of the lines will be replaced by the string 'LOST'. These lines will almost always be among the last few that you changed. You can either choose to discard the changes you made (if they are easy to redo) or to replace the few lost lines by hand.

You can get a listing of the files that are saved for you by typing:

```
hostname% vi -r
```

If there is more than one instance of a particular file saved, `vi` gives you the newest instance each time you recover it. You can thus get an older saved copy back by first recovering the newer copies.

The invocation '`vi -r`' will not always list all saved files, but they can be recovered even if they are not listed.

## Continuous Text Input — `wrapmargin`

When you are typing in large amounts of text it is convenient to have lines broken near the right margin automatically. To do this, use the *set wrapmargin* option:

```
:se wm=10<CR>
```

This rewrites words on the next line that you type past the right margin.

If `vi` breaks an input line and you wish to put it back together, you can tell it to join the lines with `J`. You can give `J` a count of the number of lines to be joined as in `3J` to join 3 lines. `vi` supplies blank space, if appropriate, at the juncture of the joined lines, and leaves the cursor at this blank space. You can delete the blank space with `x` if you don't want it.

If you want to *split* a line into two, put the cursor where you want the break, and type `rCR`.

## Features for Editing Programs

`vi` has a number of commands for editing programs. To generate correctly-indented programs, use the *autoindent* option:

```
:se ai<CR>
```

Now try opening a new line with `o`. Type a few tabs on the line and then some characters. If you type a `CR` and start another line, notice that `vi` supplies blank space at the beginning of the line to align the text of the new line with that of the previous line.

After you have started a new line, you might want to indent your current line less than the previous line. You are still in insert mode, and cannot backspace over

the automatic indentation. However, you can type CTRL-D to backtab over each level of indentation. Each time you type CTRL-D, you back up one position, normally to an eight-column boundary. You can set the number of columns that a tab shifts with the *shiftwidth* option. Try giving the command:

```
:se sw=4<CR>
```

and then experimenting with *autoindent* again.

For shifting lines in the program left and right, there are operators < and >. These shift the lines you specify right or left by one *shiftwidth*. Try << and >> which shift one line left or right, and <L and >L shifting the rest of the text left and right.

If you have a complicated expression and wish to see how the parentheses match, put the cursor at a left or right parenthesis and type %. This shows you the matching parenthesis. This works also for braces { and }, and brackets [ and ].

If you are editing C programs, you can use [[ and ]] to advance or retreat to a line starting with a {, that is, a function declaration at a time. When you use ]] with an operator, it stops after a line that starts with ); this is sometimes useful with y]].

## Filtering Portions of the Buffer

You can run system commands over portions of the buffer using the operator '!'. You can use this to sort lines in the buffer, or to reformat portions of the buffer with a pretty printer. Try typing in a list of random words, one per line and ending them with a blank line. Back up to the beginning of the list, and then give the command:

```
!)sort<CR>
```

This says to sort the next paragraph of material, and that the blank line ends a paragraph. The result is sorted text in your file.

## Commands for Editing LISP

If you are editing a LISP program, set the option *lisp* by doing:

```
:se lisp<CR>
```

This changes the ( and ) commands to move backward and forward over s-expressions. The { and } commands are like ( and ) but don't stop at atoms. Use { and } to skip to the next list, or through a comment quickly.

The *autoindent* option works differently for LISP, supplying indentation to align at the first argument to the last open list. If there is no such argument, the indent is two spaces more than the last level.

The *showmatch* option shows matching parentheses. Try setting it with:

```
:se sm<CR>
```

and then try typing a '(' *some words* and then a ')'. Notice that the cursor briefly shows the position of the '(' which matches the ')'. This happens only if the matching '(' is on the screen, and the cursor stays there for at most one second.

vi also has an operator to realign existing lines as though they had been typed in with *lisp* and *autoindent* set. This is the = operator. Try the command =% at

the beginning of a function. This realigns all the lines of the function declaration.

When you are editing LISP, the `[[` and `]]` advance and retreat to lines beginning with a `(`, and are useful for dealing with entire function definitions.

## Macros

`vi` has a parameterless macro facility you can set up so that when you type a single keystroke, `vi` will act as though you had typed some longer sequence of keys. Set this up if you find yourself repeatedly typing the same sequence of commands or text.

Briefly, there are two kinds of macros:

1. Ones where you put the macro body in a buffer register, say *x*. You can then type `@x` to invoke the macro. The `@` may be followed by another `@` to repeat the last macro.
2. You can use the `map` command from `vi` (typically in your EXINIT) with a command of the form:

```
:map lhs rhs<CR>7
```

mapping *rhs* into *lhs*. There are restrictions: *lhs* should be one keystroke (either one character or one function key) since it must be entered within one second unless *notimeout* (see the "Option Descriptions" section) is set. In that case you can type it as slowly as you wish, and `vi` will wait for you to finish before it echoes anything). The *lhs* can be no longer than ten characters, the *rhs* no longer than 100. To get a space, tab or newline into *lhs* or *rhs*, escape them with a CTRL-V. It may be necessary to double the CTRL-V if you use the `map` command inside `vi`, rather than in `ex`. You do not need to escape spaces and tabs inside the *rhs*.

Thus to make the `q` key write and exit `vi`, type:

```
:map q :wq^V^V<CR> <CR>
```

which means that whenever you type `q`, it will be as though you had typed the four characters `:wqCR`. A CTRL-V is needed because without it the CR would end the `:` command, rather than becoming part of the *map* definition. There are two CTRL-Vs because from within `vi`, you must type two CTRL-Vs to get one. The first CR is part of the *rhs*, the second terminates the `:` command.

You can delete macros with

```
:unmap lhs
```

If the *lhs* of a macro is `'#0'` through `'#9'`, this maps the particular function key instead of the two-character `'#'` sequence. So that terminals without function keys can access such definitions, the form `'#x'` will mean function key *x* on all terminals and need not be typed within one second. You can change the character `'#'` by using a macro in the usual way:

---

<sup>7</sup> *lhs* is an abbreviation for *left hand side*. *rhs* is an abbreviation for *right hand side*.

```
:map ^V^V^I #
```

to use tab, for example. This won't affect the `map` command, which still uses `#`, but just the invocation from visual mode.

The `undo` command reverses an entire macro call as a unit, if it made any changes.

Placing a `!` after the word `map` applies the mapping to input mode, rather than command mode. So, to arrange for CTRL-T to be the same as four spaces in input mode, type:

```
:map! ^T ^Vb b b b
```

where `b` represents a blank. The CTRL-V prevents the blanks from being taken as blank space between the *lhs* and *rhs*. Type simply:

```
:map!
```

to list macros that apply during input mode and

```
:map
```

to list macros that apply during command mode.

**Word Abbreviations** — `:ab`,  
`:una`

A feature similar to macros in input mode is word abbreviation. You can type a short word and have it expanded into a longer word or words with `:abbreviate (:ab)`. For example:

```
:ab foo find outer otter
```

always changes the word 'foo' into the phrase 'find outer otter'. Word abbreviation is different from macros in that only whole words are affected. If 'foo' were typed as part of a larger word, it would be left alone. Also, the partial word is echoed as it is typed. There is no need for an abbreviation to be a single keystroke, as it should be with a macro. This only operates in visual mode and uses the same syntax as the `map` command, except that there are no '`!`' forms.

Use `:unabbreviate (:una)` to turn off the abbreviation. To unabbreviate the above, for example, type:

```
:una foo
```

The `vi` editor has a number of short commands that abbreviate the longer commands we have introduced here. You can find these commands easily in the "ex Commands" section of the "ex Quick Reference". They often save a bit of typing, and you can learn them when it's convenient.

## 2.8. Nitty-gritty Details

The following presents some functional details and some `ex` commands (see the "File Manipulation Commands" section) that are particularly useful in `vi`.

## Line Representation in the Display

`vi` folds long logical lines onto many physical lines in the display. Commands that advance lines advance logical lines and skip over all the segments of a line in one motion. The command `|` moves the cursor to a specific column, and may be useful for getting near the middle of a long line to split it in half. Try `80|` on a line over 80 columns long. You can make long lines very easily by placing the cursor on the first line of two you want to join and typing `shift-J` (capital J).

`vi` only puts full lines on the display; if there is not enough room on the display to fit a logical line, the `vi` editor leaves the physical line empty, placing only an '@' on the line as a place holder. (When you delete lines on a dumb terminal, `vi` will often just clear the lines to '@' to save time rather than rewriting the rest of the screen.) You can always maximize the information on the screen with `CTRL-R`.

If you wish, you can have the editor place line numbers before each line on the display. To enable this, type the option:

```
:se nu<CR>
```

To turn it off, use the *no numbers* option:

```
:se nonu<s-2CR>
```

You can have tabs represented as `CTRL-I` (appears as `^I`) and the ends of lines indicated with '\$' by giving the *list* option:

```
:se list<CR>
```

To turn this off, use:

```
:se nolist<CR>
```

Finally, lines consisting of only the character `~` are displayed when the last line in the file is in the middle of the screen. These represent physical lines that are past the logical end of file.

## Command Counts

Most `vi` commands use a preceding count to affect their behavior in some way. The following table lists the common ways the counts are used:

New window size	: / ? [[ ]] ` `
Scroll amount	CTRL-D CTRL-U
Line/column number	z G
Repeat effect	Most of the rest

`vi` maintains a notion of the current default window size. (On terminals that run at speeds greater than 1200 baud, `vi` uses the full terminal screen. On terminals slower than 1200 baud, and most dialup lines are in this group, `vi` uses eight lines as the default window size. At 1200 baud, the default is 16 lines.)

`vi` uses the default window size when it clears and refills the screen after a search or other motion moves far from the edge of the current window. All commands that take a new window size as count often redraw the screen. If you anticipate this, but do not need as large a window as you are currently using, you may wish to change the screen size by specifying the new size before these

commands. In any case, the number of lines used on the screen will expand if you move off the top with a '-' or similar command or off the bottom with a command such as RETURN or CTRL-D. The window will revert to the last specified size the next time it is cleared and refilled, but not by a CTRL-L which just redraws the screen as it is.

The scroll commands CTRL-D and CTRL-U likewise remember the amount of scroll last specified, using half the basic window size initially. The simple insert commands use a count to specify a repetition of the inserted text. Thus 10a+---ESC inserts ten repetitions of a plus sign followed by four minus signs:

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
```

A few commands also use a preceding count as a line or column number.

Except for the few commands that ignore any counts, such as CTRL-R, the rest of the vi commands use a count to indicate a simple repetition of their effect. Thus 5w advances five words on the current line, while 5RETURN advances five lines. A very useful instance of a count as a repetition is a count given to the . command, which repeats the last changing command. If you do dw and then 3., you delete first one and then three words. You can then delete two more words with 2..

## File Manipulation Commands

The following table lists the file manipulation commands you can use when you are in vi.

Table 2-2 *File Manipulation Commands*

Command	Meaning
:w	Write back changes
:wq	Write and quit
:x	Write (if necessary) and quit (same as ZZ).
:e <i>name</i>	Edit file <i>name</i>
:e!	Re-edit, discarding changes
:e + <i>name</i>	Edit, starting at end
:e + <i>n</i>	Edit, starting at line <i>n</i>
:e #	Edit alternate file
:w <i>name</i>	Write file <i>name</i>
:w! <i>name</i>	Overwrite file <i>name</i>
:x, <i>y</i> w <i>name</i>	Write lines <i>x</i> through <i>y</i> to <i>name</i>
:r <i>name</i>	Read file <i>name</i> into buffer
:r ! <i>cmd</i>	Read output of <i>cmd</i> into buffer
:n	Edit next file in argument list
:n!	Edit next file, discarding changes to current
:n <i>args</i>	Specify new argument list
:ta <i>tag</i>	Edit file containing tag <i>tag</i> , at <i>tag</i>

A CR or ESC follows all of these commands. The most basic commands are :w and :e. End a normal editing session on a single file with a ZZ command. If you are editing for a long period of time, use the :w command occasionally after major amounts of editing, and then finish with a ZZ. When you edit more

than one file, you can finish with one with a `:w` and start editing a new file by giving a `:e` command, or set *autowrite* and use `:n file`.

If you make changes to the editor's copy of a file, but do not wish to write them back, give an `!` after the command you would otherwise use to exit without changing the file. Use this carefully.

Use the `:e` command with a `+` argument to start at the end of the file, or a `+n` argument to start at line *n*. In actuality, *n* may be any editor command not containing a space, usually a scan like `+/pat` or `+?pat`. In forming new names to the `:e` command, use the character `%` which is replaced by the current filename, or the character `#` which is replaced by the alternate filename. The alternate filename is generally the last name you typed other than the current file. Thus if you try to do a `:e` and get a diagnostic that you haven't written the file, you can give a `:w` command and then a `:e #` command to redo the previous `:e`.

You can write part of the buffer to a file by finding out the lines that bound the range to be written using CTRL-G, and giving these numbers after the `:` and before the `w`, separated by `,`s. You can also mark these lines with `m` and then use an address of the form `'x,'y` on the `w` command here.

You can read another file into the buffer after the current line by using the `:r` command. You can similarly read in the output from a command, just use `!cmd` instead of a filename.

If you wish to edit a set of files in succession, you can give all the names on the command line, and then edit each one in turn using the command `:n`. To respecify the list of files to be edited, give the `:n` command a list of filenames, or a pattern to be expanded as you would have given it on the initial `vi` command.

For editing large programs, use the `:ta` command. It utilizes a data base of function names and their locations, which can be created by programs such as *ctags*, (see the *Commands Reference Manual for the Sun Workstation*) to quickly find a function whose name you give. If the `:ta` command will require the editor to switch files, then you must `:w` or abandon any changes before switching. You can repeat the `:ta` command without any arguments to look for the same tag again.

## More about Searching for Strings

When you are searching for strings in the file with `/` and `?`, `vi` normally places you at the next or previous occurrence of the string. If you are using an operator such as `d`, `c` or `y`, then you may well wish to affect lines up to the line before the line containing the pattern. You can give a search of the form `/pat/-n` to refer to the *n*th line before the next line containing *pat*, or you can use `+` instead of `-` to refer to the lines after the one containing *pat*. If you don't give a line offset, `vi` will affect characters up to the match place, rather than whole lines; thus use `+0` to affect the line that matches.

To have `vi` ignore the case of words in searches, give the *ignorecase* option:

```
:se ic<CR>
```

To turn this off so that `vi` recognizes case again, use:

```
:se noic<CR>
```

Strings given to searches may actually be regular expressions. If you do not want or need this facility, you should put:

```
set nomagic
```

in your EXINIT. When *nomagic* is set, only the characters caret (^) and dollar sign (\$) are special in patterns. The character backslash (\) is also special with *nomagic* set. You can precede some of the normally special characters (not special in *nomagic* mode) with a backslash to enable their special properties.

It is necessary to use a backslash (\) before a slash (/) to search for a slash character in a forward scan and before a question mark (?) to search for a question mark in a backward scan. The command to search for a slash character is shown on the last line of the example below, as it would appear on your screen.

```
text text
text text text/text text text text text text text
text text text text text text text text text text text
text text? text text text text text text text text? text
text text text/text text text text text/text text
text text text text text text text text text text text
//<CR>
```

The following table gives the extended forms when *magic* is set.

Table 2-3 *Extended Pattern Matching Characters*

Character	Meaning
^	At beginning of pattern, matches beginning of line
\$	At end of pattern, matches end of line
.	Matches any character
\<	Matches the beginning of a word
\>	Matches the end of a word
[ <i>string</i> ]	Matches any single character in <i>string</i>
[^ <i>string</i> ]	Matches any single character not in <i>string</i>
[ <i>x-y</i> ]	Matches any character between <i>x</i> and <i>y</i>
*	Matches any number of the preceding pattern

If you use *nomagic* mode, use the '.', '[' and '\*' primitives with a preceding \.

## More about Input Mode

There are a number of characters to make corrections during input mode. These are summarized in the following table.

Table 2-4 *Input Mode Corrections*

Character	Meaning
CTRL-H	Deletes the last input character
CTRL-W	Deletes the last input word
erase	Your erase character, same as CTRL-H
kill	Your kill character, deletes the input on this line
\	Escapes a following CTRL-H and your erase and kill
ESC	Ends an insertion
DEL	Interrupts an insertion, terminating it abnormally
CR	Starts a new line
CTRL-D	Backtabs over <i>autoindent</i>
0CTRL-D	Kills all the <i>autoindent</i>
^CTRL-D	Same as CTRL-D, but restores indent next line
CTRL-V	Quotes the next non-printing character into the file

The most usual way of making corrections to input is to type DEL (CTRL-H on a terminal) to correct a single character, or by typing one or more CTRL-W to back over incorrect words.

Your system kill character CTRL-U (or sometimes CTRL-X) erases all the input you have given on the current line. In general, you can neither erase input back around a line boundary nor can you erase characters you did not insert with this insertion command. To make corrections on the previous line after a new line has been started, press ESC to end the insertion, move over and make the correction, and then return to where you were to continue. Use A to append at the end of the current line; this is often useful for continuing text input.

If you wish to type in your erase or kill character, say CTRL-U, you must precede it with a \, just as you would do at the normal system command level. A more general way of typing non-printing characters into the file is to precede them with a CTRL-V. The CTRL-V echoes as a ^ character on which the cursor rests. This indicates that the editor expects you to type a control character. In fact you may type any character and it will be inserted into the file at that point.<sup>8</sup>

If you are using *autoindent*, you can backtab over the indent that it supplies by typing a CTRL-D. This backs up to a *shiftwidth* boundary. This only works immediately after the supplied *autoindent*.

When you are using *autoindent* you may wish to place a label at the left margin of a line. The way to do this easily is to type a caret (CTRL-) and then CTRL-D. The editor will move the cursor to the left margin for one line, and restore the previous indent on the next. You can also type a zero (0) followed immediately

<sup>8</sup> This is not quite true. vi does not allow the NULL (CTRL-@) character to appear in files. Also the editor uses the LF (linefeed or CTRL-J) character to separate lines in the file, so it cannot appear in the middle of a line. You can insert any other character, however, if you wait for the editor to echo the ^ before you type the character. In fact, the editor treats a following letter as a request for the corresponding control character. This is the only way to type CTRL-S or CTRL-Q, since the system normally uses them to suspend and resume output and never gives them to the editor to process.

by a CTRL-D if you wish to kill all the indent and not have it come back on the next line.

## 2.9. Command and Function Reference

The following section provides abridged explanations of the `vi` and `ex` commands.

### Notation

Notation used in this section is as follows.

*[option]* Denotes optional parts of a command. Many `vi` commands have an optional count.

*[count]* Means that an optional number may precede the command to multiply or iterate the command.

*{variable item}*

Denotes parts of the command that must appear, but can take a number of different values.

*<character [-character]>*

Means that the character or one of the characters in the range described between the two angle brackets is to be typed. For example `ESC` means type the ESCAPE key. `<a-z>` means type a lowercase letter. `CTRL-<character>` means type the character as a *control* character, that is, with the CTRL key held down while simultaneously typing the specified character. Here we indicate control characters with *upper-case* letters, but `CTRL-<uppercase letter>` and `CTRL-<lowercase letter>` are equivalent. That is, `CTRL-D` is equal to `CTRL-d`. The most common character abbreviations used in this list are as follows:

Table 2-5 *Common Character Abbreviations*

Character Abbreviation	Meaning	Hexadecimal Representation
ESC	escape	0x1b
CR	carriage return, CTRL-M	0xd
<lf>	linefeed CTRL-J	0xa
<nl>	newline, CTRL-J	0xa (same as linefeed)
<bs>	backspace, CTRL-H	0x8
<tab>	tab, CTRL-I	0x9
<bell>	bell, CTRL-G	0x7
<ff>	formfeed, CTRL-L	0xc
<sp>	space	0x20
DEL	delete	0x7f

## Commands

Following are brief explanations of the `vi` commands categorized by function for easy reference.

## Entry and Exit

To use `vi` to edit a particular file, type:

```
hostname% vi filename
```

`vi` will read the file into the buffer, and place the cursor at the beginning of the first line. The first screenful of the file is displayed on the screen.

To exit from `vi`, type:

```
ZZ (or :x or :q or :q!)
```

If you are in some special mode, such as input mode or the middle of a multi-keystroke command, it may be necessary to type ESC first.

## Cursor and Page Motion

Note: You can move the cursor on your screen with the arrow keys on your workstation keyboard, the control character versions, or the `h`, `j`, `k`, and `l` keys. If you are using a terminal that does not have arrow keys, use the control character versions or the `h`, `j`, `k`, and `l` keys.

`[count]<bs>` or `[count]h` or `[count]←`

Move the cursor to the left one character. Cursor stops at the left margin of the page. `[count]` specifies the number of spaces to move.

`[count]CTRL-N` or `[count]j` or `[count]↓` or `[count]<lf>`

Move down one line. Moving off the screen scrolls the window to force a new line onto the screen. Mnemonic: Next

`[count]CTRL-P` or `[count]k` or `[count]↑`

Move up one line. Moving off the top of the screen forces new text onto the screen. Mnemonic: Previous

`[count]<sp>` or `[count]l` or `[count]→`

Move to the right one character. Cursor will not go beyond the end of the line.

`[count]–`

Move the cursor up the screen to the beginning of the next line. Scroll if necessary.

`[count]+` or `[count]CR`

Move the cursor down the screen to the beginning of the next line. Scroll up if necessary.

`[count]$`

Move the cursor to the end of the line. If there is a count, move to the end of the line *count* lines forward in the file.

`^`

Move the cursor to the beginning of the first word on the line.

`0`

Move the cursor to the left margin of the current line.

`[count]|`

Move the cursor to the column specified by the count. The default is column zero.

[count]w	Move the cursor to the beginning of the next word. If there is a count, then move forward that many words and position the cursor at the beginning of the word. Mnemonic: next-word
[count]W	Move the cursor to the beginning of the next word that follows a blank space (<sp>,<tab>, or <nl>). Ignore other punctuation.
[count]b	Move the cursor to the preceding word. Mnemonic: backup-word
[count]B	Move the cursor to the preceding word that is separated from the current word by a blank space (<sp>,<tab>, or <nl>).
[count]e	Move the cursor to the end of the current word or the end of the <i>count</i> th word hence. Mnemonic: end-of-word
[count]E	Move the cursor to the end of the current word which is delimited by blank space (<sp>,<tab>, or <nl>).
[line number]G	Move the cursor to the line specified. Of particular use are the sequences 1G and G, which move the cursor to the beginning and the end of the file respectively. Mnemonic: Go-to

Note: The next four commands (CTRL-D, CTRL-U, CTRL-F, CTRL-B) are not true motion commands, in that they cannot be used as the object of commands such as delete or change.

[count]CTRL-D	Move the cursor down in the file by <i>count</i> lines (or the last <i>count</i> if a new count isn't given). The initial default is half a page. The screen is simultaneously scrolled up. Mnemonic: Down
[count]CTRL-U	Move the cursor up in the file by <i>count</i> lines. The screen is simultaneously scrolled down. Mnemonic: Up
[count]CTRL-F	Move the cursor to the next page. A count moves that many pages. Two lines of the previous page are kept on the screen for continuity if possible. Mnemonic: Forward
[count]CTRL-B	Move the cursor to the previous page. Two lines of the current page are kept if possible. Mnemonic: Backward
[count])	Move the cursor to the beginning of the next sentence. A sentence is defined as ending with a '.', '!', or '?' followed by two spaces or a <nl>.
[count](	Move the cursor backward to the beginning of a sentence.
[count]}	Move the cursor to the beginning of the next paragraph. This command works best inside <code>nroff</code> documents. It understands the <code>nroff</code> macros in <code>-ms</code> , for which the commands <code>.IP</code> , <code>.LP</code> , <code>.PP</code> , <code>.QP</code> , as well as the <code>nroff</code> command <code>.bp</code> are considered to be paragraph delimiters. A blank line also delimits a paragraph. The <code>nroff</code> macros that it

	accepts as paragraph delimiters are adjustable. See the entry for "Paragraphs" in the "Set Commands" section.
[count]{	Move the cursor backward to the beginning of a paragraph.
}}	Move the cursor to the next 'section,' where a section is defined by the set of <code>nroff</code> macros in <code>-ms</code> , in which <code>.NH</code> , <code>.SH</code> and <code>.H</code> delimit a section. A line beginning with a <code>&lt;ff&gt;&lt;nl&gt;</code> sequence, or a line beginning with a <code>{</code> are also considered to be section delimiters. The last option makes it useful for finding the beginnings of C functions. The <code>nroff</code> macros that are used for section delimiters can be adjusted. See the "sections" entry under the heading "Set Commands".
[[	Move the cursor backward to the beginning of a section.
%	Move the cursor to the matching parenthesis or brace. This is very useful in C or lisp code. If the cursor is sitting on a <code>(</code> , <code>)</code> , <code>{</code> , or <code>}</code> , it is moved to the matching character at the other end of the section. If the cursor is not sitting on a brace or a parenthesis, <code>vi</code> searches forward on that line until it finds one and then jumps to the match mate.
[count]H	If there is no count, move the cursor to the top left position on the screen. If there is a count, then move the cursor to the beginning of the line <i>count</i> lines from the top of the screen. Mnemonic: Home
[count]L	If there is no count, move the cursor to the beginning of the last line on the screen. If there is a count, move the cursor to the beginning of the line <i>count</i> lines from the bottom of the screen. Mnemonic: Last
M	Move the cursor to the beginning of the middle line on the screen. Mnemonic: Middle
m<a-z>	<i>Mark</i> the place in the file without moving the cursor; use a character from a to z, ' <code>&lt;a-z&gt;</code> ', as the label for referring to this location in the file. See the next two commands. Mnemonic: mark Note: the <i>mark</i> command is not a motion and cannot be used as the target of commands such as <i>delete</i> .
^<a-z>	Move the cursor to the beginning of the line that is marked with the label ' <code>&lt;a-z&gt;</code> '.
`<a-z>	Move the cursor to the exact position on the line that was marked with the label ' <code>&lt;a-z&gt;</code> '.
''	Move the cursor back to the beginning of the line where it was before the last <i>non-relative</i> move. A non-relative move is something such as searching or jumping to a specific line in the file, rather than moving the cursor or scrolling the screen.

`` Move the cursor back to the exact spot on the line where it was located before the last non-relative move.

## Searches

The following commands search for items in a file.

[count]f{chr}	Search forward on the line for the next or <i>count</i> th occurrence of the character <i>chr</i> . The cursor is placed <i>at</i> the character of interest. Mnemonic: find character
[count]F{chr}	Search backward on the line for the next or <i>count</i> th occurrence of the character <i>chr</i> . The cursor is placed <i>at</i> the character of interest.
[count]t{chr}	Search forward on the line for the next or <i>count</i> th occurrence of the character <i>chr</i> . The cursor is placed <i>just preceding</i> the character of interest. Mnemonic: move cursor up to character
[count]T{chr}	Search backward on the line for the next or <i>count</i> th occurrence of the character <i>chr</i> . The cursor is placed <i>just preceding</i> the character of interest.
[count];	Repeat the last f, F, t or T command in the same search direction.
[count],	Repeat the last f, F, t or T command, but in the opposite search direction. This is useful if you overshoot what you are looking for.
[count]/[string]/<nl>	Search forward for the next occurrence of 'string'. Wraparound at the end of the file does occur. The final / is not required.
[count]?[string]?<nl>	Search backward for the next occurrence of 'string'. If a count is specified, the count becomes the new window size. Wraparound at the beginning of the file does occur. The final ? is not required.
n	Repeat the last /[string]/ or ?[string]? search. Mnemonic: next occurrence.
N	Repeat the last /[string]/ or ?[string]? search, but in the reverse direction.
:g/[string]/[editor command]<nl>	Using the : syntax, it is possible to do global searches like you can in the ed editor.

## Text Insertion

The following commands insert text. Terminate all multi-character text insertions with an ESC character. You can always *undo* the last change by typing a `u`. The text insert in insertion mode can contain newlines.

- `a{text}<esc>` Insert text immediately following the cursor position.  
Mnemonic: **append**
- `A{text}<esc>` Insert text at the end of the current line. Mnemonic: **Append**
- `i{text}<esc>` Insert text immediately preceding the cursor position.  
Mnemonic: **insert**
- `I{text}<esc>` Insert text at the beginning of the current line.
- `o{text}<esc>` Insert a new line after the line on which the cursor appears and insert text there. Mnemonic: **open new line**
- `O{text}<esc>` Insert a new line preceding the line on which the cursor appears and insert text there.

## Text Deletion

The following commands delete text in various ways. You can always *undo* changes by typing the `u` command.

- `[count]x` Delete the character or characters starting at the cursor position.
- `[count]X` Delete the character or characters starting at the character preceding the cursor position.
- `D` Delete the remainder of the line starting at the cursor.  
Mnemonic: **Delete the rest of line**
- `[count]d{motion}` Delete one or more occurrences of the specified motion. You can use any motion here described in the sections "Low Level Character Motions" and "Higher Level Text Objects". You can repeat the `d` (such as `[count]dd`) to delete *count* lines.

## Text Replacement

Use the following commands to simultaneously delete and insert new text. You can *undo* all such actions by typing `u` following the command.

- `r<chr>` Replace the character at the current cursor position with `<chr>`. This is a one-character replacement. No ESC is required for termination. Mnemonic: **replace character**
- `R{text}<esc>` Start overlaying the characters on the screen with whatever you type. It does not stop until you type an ESC.
- `[count]s{text}<esc>` Substitute for *count* characters beginning at the current cursor position. A '\$' appears at the position in the text where the *count*th character appears so you will know how much you are erasing. Mnemonic: **substitute**
- `[count]S{text}<esc>` Substitute for the entire current line or lines. If you do not

give a count, a '\$' appears at the end of the current line. If you give a count of more than 1, all the lines to be replaced are deleted before the insertion begins.

`[count]c{motion}{text}<esc>`

Change the specified *motion* by replacing it with the insertion text. A '\$' appears at the end of the last item that is being deleted unless the deletion involves whole lines. Motions can be any motion from the sections "Low Level Character Motions" and "Higher Level Text Objects". Repeat the `c` (such as `[count]cc`) to change *count* lines.

## Moving Text

You can move chunks of text around in a number of ways with `vi`. There are nine buffers into which each piece of text deleted or *yanked* is put in addition to the *undo* buffer. The most recent deletion or yank is in the *undo* buffer and also usually in buffer 1, the next most recent in buffer 2, and so forth. Each new deletion pushes down all the older deletions. Deletions older than 9 disappear. There is also a set of named registers, a-z, into which text can optionally be placed. If you precede any delete or replacement type command by "`<a-z>`", that named buffer will contain the text deleted after the command is executed. For example, "`a3dd`" deletes three lines starting at the current line and puts them in buffer "a". Referring to an upper-case letter as a buffer name (A-Z) is the same as referring to the lower-case letter, except that text placed in such a buffer is appended to it instead of replacing it. There are two more basic commands and some variations useful in getting and putting text into a file.

`["<a-z>][count]y{motion}`

Yank the specified item or *count* items and put in the *undo* buffer or the specified buffer. The variety of *items* that you can yank is the same as those that you can delete with the `d` command or changed with the `c` command. In the same way that `dd` means delete the current line and `cc` means replace the current line, `yy` means yank the current line.

`["<a-z>][count]Y`

Yank the current line or the *count* lines starting from the current line. If no buffer is specified, they will go into the *undo* buffer, like any delete would. It is equivalent to `yy`. Mnemonic: Yank

`["<a-z>]p`

Put *undo* buffer or the specified buffer down *after* the cursor. If you yanked or deleted whole lines into the buffer, they are put down on the line following the line the cursor is on. If you deleted something else, like a word or sentence, it is inserted immediately following the cursor. Mnemonic: put buffer

Note that text in the named buffers remains there when you start editing a new file with the `:e fileCR` command. Since this is so, it is possible to copy or delete text from one file and carry it over to another file in the buffers. However, the *undo* buffer and the ability to *undo* are lost when changing

files.

- [ "<a-z>]P Put *undo* buffer or the specified buffer down *before* the cursor. If you yanked or deleted whole lines into the buffer, they are put down on the line preceding the line the cursor is on. If you deleted something else, like a word or sentence, it is inserted immediately preceding the cursor.
- [count]>{motion} The shift operator right shifts all the text from the line on which the cursor is located to the line where the *motion* is located. The text is shifted by one *shiftwidth*. (See the "Terminal Information" section.) >> means right shift the current line or lines.
- [count]<{motion} The shift operator left shifts all the text from the line on which the cursor is located to the line where the *item* is located. The text is shifted by one *shiftwidth*. (See the section on "Terminal Information".) << means left shift the current line or lines. Once the line has reached the left margin, it is not affected further.
- [count]={motion} Prettyprints the indicated area according to LISP conventions. The area should be a LISP s-expression.

## Miscellaneous Commands

A number of useful miscellaneous `vi` commands follow:

- `ZZ`, Exit from `vi`. If any changes have been made, the file is written out. Then you are returned to the shell.
- `CTRL-L` Redraw the current screen. This is useful if messages from a background process are displayed on the screen, if someone 'writes' to you while you are using `vi` or if for any reason garbage gets onto the screen.
- `CTRL-R` On dumb terminals, those not having the 'delete line' function (the vt100 for example), `vi` saves redrawing the screen when you delete a line by just marking the line with an '@' at the beginning and blanking the line. If you want to actually get rid of the lines marked with '@' and see what the page looks like, type a `CTRL-R`.
- `.` 'Dot' repeats the last text modifying command. You can type a command once and then move to another place and repeat it by just typing '.'.
- `u` Undo the last command that changed the buffer. Perhaps the most important command in the editor. Mnemonic: `undo`
- `U` Undo all the text modifying commands performed on the current line since the last time you moved onto it.
- [count]`J` Join the current line and the following line. The `<nl>` is deleted and the two lines joined, usually with a space between the end of the first line and the beginning of what

was the second line. If the first line ended with a ‘period’, two spaces are inserted. A count joins the next *count* lines. Mnemonic: Join lines

**Q** Switch to *ex* editing mode. In this mode *vi* behaves very much like *ed*. The editor in this mode operates on single lines normally and does not attempt to keep the ‘window’ up to date. Once in this mode you can also switch to the *open* mode of editing by entering the command [*line number*]open<nl>. It is similar to the normal visual mode except the window is only *one* line long. Mnemonic: Quit visual mode

**CTRL-]** An abbreviation for a *tag* command. The cursor should be positioned at the beginning of a word. That word is taken as a tag name, and the tag with that name is found as if it had been typed in a *:tag* command.

[*count*]!*{motion}*{*Sun cmd*}<nl>  
Any Sun system filter (that is, a command that reads the standard input and outputs something to the standard output) can be sent a section of the current file and have the output of the command replace the original text. Useful examples are programs like *cb*, *sort*, and *nroff*. For instance, using *sort* you can sort a section of the current file into a new list. Using *!!* means take a line or lines starting at the line the cursor is currently on and pass them to the Sun system command. Note: To escape to the shell for just one command, use *:!{cmd}<nl>* (see the "High Level Commands" section).

**z{count}<nl>** Reset the current window size to *count* lines and redraw the screen.

## Special Insert Characters

Following are some characters that have special meanings during insert mode.

**CTRL-V** During inserts, typing a CTRL-V quotes control characters into the file. Any character typed after the CTRL-V is inserted into the file.

**[^]CTRL-D** CTRL-D without any argument backs up one *shiftwidth*. Use this to remove indentation that was inserted by the *autoindent* feature. Typing ^CTRL-D temporarily removes all the autoindentation, thus placing the cursor at the left margin. On the next line, the previous indent level is restored. This is useful for putting ‘labels’ at the left margin. OCTRL-D removes all autoindents and keeps it that way. Thus the cursor moves to the left margin and stays there on successive lines until you type TABs. As with the TAB, the CTRL-D is effective only before you type any other ‘non-autoindent’ controlling characters. Mnemonic: Delete a shiftwidth

CTRL-W	If the cursor is sitting on a word, CTRL-W moves the cursor back to the beginning of the word, erasing the word from the insert. Mnemonic: erase <b>W</b> ord
<bs>	The backspace always serves as an erase during insert modes in addition to your normal 'erase' character. To insert a <bs> into your file, quote it with the CTRL-V.

## : Commands

Typing a colon (:) during command mode puts the cursor at the bottom on the screen in preparation for a command. In the : mode, you can give `vi` most `ex` commands. You can also exit from `vi` or switch to different files from this mode. Terminate all commands of this variety by a <nl>, <cr>, or ESC.

<code>:w[!]</code> [file]	Write out the current text to the disk. It is written to the file you are editing unless you supply <i>file</i> . If <i>file</i> is supplied, the write is directed to that file instead. If that file already exists, <code>vi</code> does not write unless you use the '!' indicating you <i>really</i> want to write over the older copy of the file.
<code>:q[!]</code>	Exit from <code>vi</code> . If you have modified the file you are currently looking at and haven't written it out, <code>vi</code> refuses to exit unless you type the !.
<code>:e[!]</code> [+ <i>cmd</i> ]] [file]	Start editing a new file called <i>filename</i> or start editing the current file over again. The command <code>:e!</code> says 'ignore the changes I've made to this file and start over from the beginning'. Use it if you really mess up the file. The optional '+' says instead of starting at the beginning, start at the 'end', or, if you supply <i>cmd</i> , execute <i>cmd</i> first. Use this where <i>cmd</i> is <i>n</i> (any integer) that starts at line number <i>n</i> , and <i>/text</i> , searches for 'text' and starts at the line where it is found.
CTRL-^	Switch back to the place in the previous file that you were editing with <code>vi</code> , before you switched to the current file. (Same as <code>:e #</code> on the command line.)
<code>:n[!]</code>	Start editing the next file in the argument list. Since you can call <code>vi</code> with multiple filenames, the <code>:n</code> command tells it to stop work on the current file and switch to the next file. If you have modified the current file, it has to be written out before the <code>:n</code> will work or else you must use '!', which discards the changes you made to the current file.
<code>:n[!]</code> file [file file ...]	Replace the current argument list with a new list of files and start editing the first file in this new list.
<code>:r</code> file	Read in a copy of <i>file</i> on the line after the cursor.
<code>:r !cmd</code>	Execute the <i>cmd</i> and take its output and put it into the file after the current line.

`!cmd` Execute any system shell command.

`:ta[!] tag` vi looks in the file named *tags* in the current directory. *tags* is a file of lines in the format:

`tag filename vi-search-command`

If vi finds the tag you specified in the `:ta` command, it stops editing the current file if necessary. If the current file is up to date on the disk, it switches to the file specified and uses the search pattern specified to find the 'tagged' item of interest. Use this when editing multi-file C programs such as the operating system. There is a program called *ctags* which generates an appropriate *tags* file for C and f77 programs so that by saying `:ta function<nl>` you can switch to that function. It can also be useful when editing multi-file documents, though the *tags* file has to be generated manually in this case.

## Set Commands

vi has a number of internal variables and switches you can set to achieve special affects. These options come in three forms: switches that toggle off or on, options that require a numeric value, and options that require an alphanumeric string value. Set the toggle options by a command of the form:

```
:set option<nl>
```

and turn off the toggle options with the command:

```
:set nooption<nl>
```

To set commands requiring a value, use a command of the form:

```
:set option=value<nl>
```

To display the value of a specific option, type:

```
:set option?<nl>
```

To display only those that you have changed, type:

```
:set<nl>
```

and to display the long table of all the settable parameters and their current values, type:

```
:set all<nl>
```

Most of the options have a long form and an abbreviation. Both are described in the following list as well as the normal default value.

To use values other than the default every time you enter vi, place the appropriate *set* command in EXINIT in your environment, such as:

```
setenv EXINIT 'set ai aw terse sh=/bin/csh'
```

or

```
EXINIT='set ai aw terse sh=/bin/csh'
export EXINIT
```

for *cs*h and *sh*, respectively. Place these in your *.login* or *.profile* file in your home directory.

autoindent ai	Default: noai Type: toggle When in autoindent mode, <i>vi</i> helps you indent code by starting each line in the same column as the preceding line. Tabbing to the right with <tab> or CTRL-T moves this boundary to the right; to move it to the left, use CTRL-D.
autoprint ap	Default: ap Type: toggle Displays the current line after each <i>ex</i> text modifying command. Not of much interest in the normal <i>vi</i> visual mode.
autowrite aw	Default: noaw type: toggle Does an automatic write if there are unsaved changes before certain commands that change files or otherwise interact with the outside world are executed. These commands are <i>!</i> , <i>:tag</i> , <i>:next</i> , <i>:rewind</i> , CTRL-^, and CTRL-].
beautify bf	Default: nobf Type: toggle Discards all control characters except <tab>, <nl>, and <ff>.
directory dir	Default: dir= <i>/tmp</i> Type: string This is the directory in which <i>vi</i> puts its temporary file.
errorbells eb	Default: noeb Type: toggle Error messages are preceded by a <bell>. Sun Workstations not equipped with speakers flash instead of beeping.
hardtabs ht	Default: hardtabs=8 Type: numeric This option contains the value of hardware tabs in your terminal, or of software tabs expanded by the Sun system.
ignorecase ic	Default: noic Type: toggle Map all upper-case characters to lower case in regular expression matching.
lisp	Default: nolisp Type: toggle Autoindent for LISP code. The commands (, ), [, and ] are modified appropriately to affect s-expressions and functions.
list	Default: nolist Type: toggle Show the <tab> and <nl> characters visually on all displayed lines.
magic	Default: magic Type: toggle Enable the metacharacters for matching. These include ., *, <, >, [ <i>string</i> ], [^ <i>string</i> ], and [<chr>-<chr>].
number nu	Default: nonu Type: toggle Display each line with its line number.

open	<p>Default: open Type: toggle</p> <p>When set, prevents entering open or visual modes from <code>ex</code> or <code>edit</code>. Not of interest from <code>vi</code>.</p>
optimize opt	<p>Default: opt Type: toggle</p> <p>Useful only when using the <code>ex</code> capabilities. This option prevents automatic <code>&lt;cr&gt;</code>s from taking place, and speeds up output of indented lines, at the expense of losing typeahead on some versions of the operating system.</p>
paragraphs para	<p>Default: para=IPLPPPQPP bp Type: string</p> <p>Each pair of characters in the string indicates <code>nroff</code> macros to be treated as the beginning of a paragraph for the <code>{</code> and <code>}</code> commands. The default string is for the <code>-ms</code> macros. To indicate one-letter <code>nroff</code> macros, such as <code>.P</code> or <code>.H</code>, insert a space for the second character position. For example:</p> <pre>:set paragraphs=PPH\ bp&lt;nl&gt;</pre> <p>causes <code>vi</code> to consider <code>.PP</code>, <code>.H</code> and <code>.bp</code> as paragraph delimiters.</p>
prompt	<p>Default: prompt Type: toggle</p> <p>In <code>ex</code> command mode the prompt character <code>:</code> is displayed when <code>ex</code> is waiting for a command. This is not of interest from <code>vi</code>.</p>
redraw	<p>Default: noredraw Type: toggle</p> <p>On dumb terminals, force the screen to always be up to date by sending great amounts of output. Useful only at high speeds.</p>
report	<p>Default: report=5 Type: numeric</p> <p>Set the threshold for the number of lines modified. When more than this number of lines is modified, removed, or yanked, <code>vi</code> reports the number of lines changed at the bottom of the screen.</p>
scroll	<p>Default: scroll={1/2 window} Type: numeric</p> <p>This is the number of lines that the screen scrolls up or down when using the <code>CTRL-U</code> and <code>CTRL-D</code> commands.</p>
sections	<p>Default: sections=SHNHH HU Type: string</p> <p>Each two-character pair of this string specifies <code>nroff</code> macro names that are to be treated as the beginning of a section by the <code>]</code> and <code>[</code> commands. The default string is for the <code>-ms</code> macros. To enter one-letter <code>nroff</code> macros, use a quoted space as the second character. See the "Paragraphs" entry for a fuller explanation.</p>
shell sh	<p>Default: sh=from environment SHELL or /bin/sh Type: string</p> <p>Specify the name of the <code>sh</code> to be used for 'escaped'</p>

	commands.
shiftwidth sw	Default: sw=8 Type: numeric Specify the number of spaces that a CTRL-T or CTRL-D will move over for indenting, and the amount that < and > will shift by.
showmatch sm	Default: nosm Type: toggle When a ) or } is typed, show the matching ( or { by moving the cursor to it for one second if it is on the current screen.
slowopen slow	Default: terminal dependent Type: toggle Prevent updating the screen some of the time to improve speed on terminals that are slow and dumb.
tabstop ts	Default: ts=8 Type: numeric <tab>s are expanded to boundaries that are multiples of this value.
taglength tl	Default: tl=0 Type: numeric If nonzero, tag names are only significant to this many characters.
term	Default: (from environment TERM, else dumb) Type: string This is the terminal and controls the visual displays. It cannot be changed when in visual mode; you have to type a Q to change to command mode, type a set term command, and enter vi to get back into visual. Or exit from vi, fix \$TERM, and re-enter. The definitions that drive a particular terminal type are in the file /etc/termcap.
terse	Default: terse Type: toggle When set, the error diagnostics are short.
warn	Default: warn Type: toggle Warns if you try to escape to the shell without writing out the current changes.
window	Default: window={8 at 600 baud or less, 16 at 1200 baud, and screen size - 1 at 2400 baud or more} Type: numeric Specify the number of lines in the window whenever vi must redraw an entire screen. It is useful to make this size smaller if you are on a slow line.
w300, w1200, w9600	Set the window, but only within the corresponding speed ranges. They are useful in an EXINIT to fine tune window sizes. For example, <pre>set w300=4 w1200=12</pre> produces a four-line window at speeds up to 600 baud, a 12-line window at 1200 baud, and a full-screen window (the default) at over 1200 baud.

wrapscan ws	Default: ws Type: toggle Searches will wrap around the end of the file when is option is set. When it is off, the search will terminate when it reaches the end or the beginning of the file.
wrapmargin wm	Default: wm=0 Type: numeric vi automatically inserts a <nl> when it finds a natural break point (usually a <sp> between words) that occurs within <i>wm</i> spaces of the right margin. Therefore with 'wm=0', the option is off. Setting it to 10 means that any time you are within 10 spaces of the right margin, vi looks for a <sp> or <tab> that it can replace with a <nl>. This is convenient if you forget to look at the screen while you type. If you go past the margin (even in the middle of a word), the entire word is erased and rewritten on the next line.
writeany wa	Default: nowa Type: toggle vi normally makes a number of checks before it writes out a file. This prevents you from inadvertently destroying a file. When the <i>writeany</i> option is enabled, vi no longer makes these checks.

## Character Functions

This section describes how the editor uses each character. The characters are presented in their order in the ASCII character set: control characters come first, then most special characters, the digits, upper-, and finally lower-case characters.

For each character we list its meaning as a command and its meaning (if any) during insert mode.

CTRL-@	Not a command character. If typed as the first character of an insertion, it is replaced with the last text inserted, and the insert terminates. Only 128 characters are saved from the last insert; if more characters were inserted the mechanism is not available. A CTRL-@ cannot be part of the file due to the editor implementation.
CTRL-A	Unused.
CTRL-B	Scroll backward one window. A count specifies repetition. The top two lines in the window before typing CTRL-B appear as the bottom two lines of the next window.
CTRL-C	Unused.
CTRL-D	As a command, scrolls down a half window of text. A count gives the number of (logical) lines to scroll, and is remembered for future CTRL-D and CTRL-U commands. During an insert, CTRL-D backtabs over <i>autoindent</i> blank space at the beginning of a line. This blank space cannot be backspaced over.
CTRL-E	Exposes one more line below the current screen in the file, leaving the cursor where it is if possible.

CTRL-F	Move forward one window. A count specifies repetition. The bottom two lines in the window before typing CTRL-F appear as the top two lines of the next window.
CTRL-G	Equivalent to :fCR. These commands display the current file, a message if the file has been modified, the line number of the line the cursor is on, the total number of lines in the file, and the percentage of the way through the file that the current line is.
CTRL-H (BS)	Same as ← (see h). During an insert, CTRL-H eliminates the last input character, backing over it but not erasing it; the character remains so you can see what you typed if you wish to type something only slightly different.
CTRL-I (TAB)	Not a command character. When inserted it prints as some number of spaces. When the cursor is at a tab character, it rests at the last of the spaces that represent the tab. The <i>tabstop</i> option controls the spacing of tabstops.
CTRL-J (LF)	Same as ↓ (see j).
CTRL-K	Unused.
CTRL-L	The ASCII formfeed character, that clears and redraws the screen. This is useful after a transmission error, if characters typed by a program other than the editor scramble the screen, or after output is stopped by an interrupt.
CTRL-M (CR)	A carriage return advances to the next line, at the first non-blank position in the line. Given a count, it advances that many lines. During an insert, a CR causes the insert to continue onto another line.
CTRL-N	Same as ↓ (see j).
CTRL-O	Unused.
CTRL-P	Same as ↑ (see k).
CTRL-Q	Not a command character. In input mode, CTRL-Q quotes the next character, the same as CTRL-V, except that some teletype drivers will eat the CTRL-Q so that <i>vi</i> never sees it. Resumes operation suspended by CTRL-S.
CTRL-R	Redraws the current screen, eliminating logical lines not corresponding to physical lines (lines with only a single @ character on them). On hardcopy terminals in <i>open</i> mode, retypes the current line.
CTRL-S	Some teletype drivers use CTRL-S to suspend output until CTRL-Q is pressed. Unused.
CTRL-T	Not a command character. During an insert with <i>autoindent</i> set and at the beginning of the line, inserts <i>shiftwidth</i> blank space.

CTRL-U	Scrolls the screen up half a window, the reverse of CTRL-D, which scrolls down. Counts work as they do for CTRL-D, and the previous scroll amount is common to both CTRL-D and CTRL-U. On a dumb terminal, CTRL-U will often necessitate clearing and redrawing the screen further back in the file.
CTRL-V	Not a command character. In input mode, quotes the next character so that it is possible to insert non-printing and special characters into the file.
CTRL-W	Not a command character. During an insert, backs up as <code>b</code> does in command mode; the deleted characters remain on the display (see CTRL-H).
CTRL-X	Unused.
CTRL-Y	Exposes one more line above the current screen, leaving the cursor where it is if possible. (No mnemonic value for this key; CTRL-Y is the reverse of CTRL-E).
CTRL-Z	Stops the editor, exiting to the top level shell. Same as <code>:stopCR</code> .
CTRL-[ (ESC)	Cancels a partially-formed command, such as <code>a z</code> when no following character has yet been given; terminates inputs on the last line (read by commands such as <code>: /</code> and <code>?</code> ); ends insertions of new text into the buffer. If an ESC is given when quiescent in command state, the editor flashes the screen or rings the bell. You can thus type ESC if you don't know what is happening till the editor flashes the screen. If you don't know if you are in insert mode, you can type <code>ESCa</code> , and then material to be input; the material is inserted correctly whether or not you were in insert mode when you started.
CTRL^	Unused.
CTRL_]	Searches for the word which is after the cursor as a tag. Equivalent to typing <code>:ta</code> , this word, and then a CR. Mnemonically, this command is 'go right to'.
CTRL^	Equivalent to <code>:e #CR</code> , returning to the previous position in the last-edited file, or editing a file that you specified if you got a <code>No write since last change diagnostic</code> and do not want to have to type the filename again. You have to do a <code>:w</code> before CTRL^ will work in this case. If you do not wish to write the file you should do <code>:e! #CR</code> instead.
CTRL_	Unused. Reserved as the command character for the Tektronix 4025 and 4027 terminal.
SPACE	Same as <code>→</code> (see 1).
!	An operator that processes lines from the buffer with reformatting commands. Follow <code>!</code> with the object to be processed, and then the command name terminated by CR. Doubling <code>!</code>

and preceding it by a count filters the count lines; otherwise the count is passed on to the object after the `!`. Thus `2!}fmtCR` reformats the next two paragraphs by running them through the program `fmt`. If you are working on LISP, the command `!%grindCR`, given at the beginning of a function, will run the text of the function through the LISP grinder. (The `grind` command may not be present at all installations.) To read the output of a command into the buffer, use `:rcommand`. To simply execute a command, [from `vi`], use `!:command`.

- " Precedes a named buffer specification. There are named buffers 1–9 used for saving deleted text and named buffers a–z into which you can place text.
- # The macro character which, when followed by a number, will substitute for a function key on terminals without function keys. In input mode, if this is your erase character, it will delete the last character you typed in input mode, and must be preceded with a `\` to insert it, since it normally backs over the last input character you gave.
- \$ Moves to the end of the current line. If you `:se listCR`, the end of each line is indicated by showing a `$` after the end of the displayed text in the line. Given a count, advances to the count'th following end of line; thus `2$` advances to the end of the next line.
- % Moves to the parenthesis or brace `{ }` that balances the parenthesis or brace at the current cursor position.
- & A synonym for `:&CR`, by analogy with the `ex &` command. When followed by a `'`, returns to the previous context at the beginning of a line. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter a–z, returns to the line that was marked with this letter with a `m` command, at the first non-blank character in the line. When used with an operator such as `d`, the operation takes place over complete lines; if you use ```, the operation takes place from the exact marked place to the current cursor position within the line.
- ( Retreats to the beginning of a sentence, or to the beginning of a LISP s-expression if the *lisp* option is set. A sentence ends at a `.`, `!`, or `?` and is followed by either the end of a line or by two spaces. Any number of closing `)`, `]`, `"`, and `'` characters may appear after the `.`, `!`, or `?`, and before the spaces or end of line. Sentences also begin at paragraph and section boundaries (see the `"{"` and `"["` entries below). A count advances that many sentences.

- ) Advances to the beginning of a sentence. A count repeats the effect. See ( above for the definition of a sentence.
- \* Unused.
- + Same as CR when used as a command.
- , Reverse of the last f, F, t, or T command, looking the other way in the current line. Especially useful after typing too many ; characters. A count repeats the search.
- Retreats to the previous line at the first non-blank character. This is the inverse of + and RETURN. If the line moved to is not on the screen, the screen is scrolled, or cleared and redrawn if scrolling is not possible. If a large amount of scrolling is required, the screen is also cleared and redrawn, with the current line at the center.
- . Repeats the last command that changed the buffer. Especially useful when deleting words or lines; you can delete some words or lines and then type . to delete more words or lines. Given a count, it passes it on to the command being repeated. Thus after a 2dw, 3. deletes three words.
- / Reads a string from the last line on the screen, and scans forward for the next occurrence of this string. The normal input editing sequences may be used during the input on the bottom line; an ESC returns to command state without ever searching. The search begins when you type CR to terminate the pattern; the cursor moves to the beginning of the last line to indicate that the search is in progress; you can then terminate the search with a CTRL-C (or DEL or RUB), or by backspacing when at the beginning of the bottom line, returning the cursor to its initial position. Searches normally wrap end-around to find a string anywhere in the buffer.
- When used with an operator, the enclosed region is normally affected. By mentioning an offset from the line matched by the pattern, you can affect whole lines. To do this, give a pattern with a closing / and then an offset +n or -n.
- To include the character / in the search string, you must escape it with a preceding \. A ^ at the beginning of the pattern forces the match to occur at the beginning of a line only; this may speed the search. A \$ at the end of the pattern forces the match to occur at the end of a line only. More extended pattern matching is available. Unless you set nomagic in your .login file (\*?\*), you will have to precede the characters ., [, \*, and ~ in the search pattern with a \ to get them to work as you would naively expect.
- 0 Moves to the first character on the current line. Also used, in forming numbers, after an initial 1-9.

- 1-9 Used to form numeric arguments to commands.
- : A prefix to a set of commands for file and option manipulation and escapes to the system. Input is given on the bottom line and terminated with a CR, and the command is then executed. You can return to where you were by typing ESC or DEL if you type : accidentally.
- ; Repeats the last single character find that used f, F, t, or T. A count iterates the basic scan.
- < An operator that shifts lines left one *shiftwidth*, normally 8 spaces. Like all operators, affects lines when repeated, as in <<. Counts are passed through to the basic object, thus 3<< shifts three lines.
- = Reindents line for LISP, as though they were typed in with *lisp* and *autoindent* set.
- > An operator that shifts lines right one *shiftwidth*, normally 8 spaces. Affects lines when repeated as in >>. Counts repeat the basic object.
- ? Scans backward, the opposite of /. See the / description above for details on scanning.
- @ A macro character. If this is your kill character, you must escape it with a \ to type it in during input mode, as it normally backs over the input you have given on the current line.
- A Appends at the end of line; a synonym for \$a.
- B Backs up a word, where words are composed of non-blank sequences, placing the cursor at the beginning of the word. A count repeats the effect.
- C Changes the rest of the text on the current line; a synonym for c\$.
- D Deletes the rest of the text on the current line; a synonym for d\$.
- E Moves forward to the end of a word, defined as blanks and non-blanks, like B and W. A count repeats the effect.
- F Finds a single following character backward in the current line. A count repeats this search that many times.
- G Goes to the line number given as preceding argument, or to the end of the file if you do not give a preceding count. The screen is redrawn with the new current line in the center if necessary.
- H Home arrow. Homes the cursor to the top line on the screen. If a count is given, the cursor is moved to the count'th line on the screen. In any case the cursor is moved to the first non-

- blank character on the line. If used as the target of an operator, full lines are affected.
- I** Inserts at the beginning of a line; a synonym for CTRL-i.
- J** Joins together lines, supplying appropriate blank space: one space between words, two spaces after a '.', and no spaces at all if the first character of the joined on line is ). A count causes that many lines to be joined rather than the default two.
- K** Unused.
- L** Moves the cursor to the first non-blank character of the last line on the screen. With a count, to the first non-blank of the count'th line from the bottom. Operators affect whole lines when used with L.
- M** Moves the cursor to the middle line on the screen, at the first non-blank position on the line.
- N** Scans for the next match of the last pattern given to / or ?, but in the reverse direction; this is the reverse of n.
- O** Opens a new line above the current line and inputs text there up to an ESC. A count can be used on dumb terminals to specify a number of lines to be opened; this is generally obsolete, as the *slowopen* option works better.
- P** Puts the last deleted text back before/above the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines. Otherwise the text is inserted between the characters before and at the cursor. May be preceded by a named buffer specification "x to retrieve the contents of the buffer; buffers 1-9 contain deleted material, buffers a-z are available for general use.
- Q** Quits from vi to ex command mode. In this mode, whole lines form commands, ending with a RETURN. You can give all the : commands; the editor supplies the : as a prompt.
- R** Replaces characters on the screen with characters you type (overlay fashion). Terminates with an ESC.
- S** Changes whole lines, a synonym for cc. A count substitutes for that many lines. The lines are saved in the numeric buffers, and erased on the screen before the substitution begins.
- T** Takes a single following character, locates the character before the cursor in the current line, and places the cursor just after that character. A count repeats the effect. Most useful with operators such as d.
- U** Restores the current line to its state before you started changing it.

- V Unused.
- W Moves forward to the beginning of a word in the current line, where words are defined as sequences of blank/non-blank characters. A count repeats the effect.
- X Deletes the character before the cursor. A count repeats the effect, but only characters on the current line are deleted.
- Y Yanks a copy of the current line into the unnamed buffer, to be put back by a later `p` or `P`; a very useful synonym for `yy`. A count yanks that many lines. May be preceded by a buffer name to put lines in that buffer.
- ZZ Exits the editor. (Same as `:xCR`.) If any changes have been made, the buffer is written out to the current file. Then the editor quits.
- [[ Backs up to the previous section boundary. A section begins at each macro in the *sections* option, normally a `.NH` or `.SH` and also at lines that start with a formfeed CTRL-L. Lines beginning with `{` also stop `[[`; this makes it useful for looking backward, a function at a time, in C programs. If the *lisp* option is set, stops at each `(` at the beginning of a line, and is thus useful for moving backward at the top level LISP objects.
- \ Unused.
- ]] Forward to a section boundary; see `[[` for a definition.
- ^ Moves to the first non-blank position on the current line.
- \_ Unused.
- ` When followed by a ``` returns to the previous context. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter `a-z`, returns to the position that was marked with this letter with an `m` command. When used with an operator such as `d`, the operation takes place from the exact marked place to the current position within the line; if you use ```, the operation takes place over complete lines.
- a Appends arbitrary text after the current cursor position; the insert can continue onto multiple lines by using RETURN within the insert. A count causes the inserted text to be replicated, but only if the inserted text is all on one line. Terminate the insertion with an ESC.
- b Backs up to the beginning of a word in the current line. A word is a sequence of alphanumerics, or a sequence of special characters. A count repeats the effect.
- c An operator that changes the following object, replacing it with the following input text up to an ESC. If more than part

of a single line is affected, the text that is changed is saved in the numeric named buffers. If only part of the current line is affected, the last character to be changed away is marked with a \$. A count causes that many objects to be affected, thus both `3c)` and `c3)` change the following three sentences.

- d** An operator that deletes the following object. If more than part of a line is affected, the text is saved in the numeric buffers. A count causes that many objects to be affected; thus `3dw` is the same as `d3w`.
- e** Advances to the end of the next word, defined as for `b` and `w`. A count repeats the effect.
- f** Finds the first instance of the next character following the cursor on the current line. A count repeats the find.
- g** Unused.
- h** Left arrow. Moves the cursor one character to the left. Like the other arrow keys, either `h`, the left arrow key, or one of the synonyms, CTRL-H has the same effect. A count repeats the effect.
- i** Inserts text before the cursor.
- j** Down arrow. Moves the cursor one line down in the same column. If the position does not exist, `vi` comes as close as possible to the same column. Synonyms include CTRL-J (linefeed) and CTRL-N.
- k** Up arrow. Moves the cursor up one line. CTRL-P is a synonym.
- l** Right arrow. Moves the cursor one character to the right. SPACE is a synonym.
- m** Marks the current position of the cursor in the mark register that is specified by the next character `a-z`. Return to this position or use with an operator using `` or `.`.
- n** Repeats the last `/` or `?` scanning commands.
- o** Opens new lines below the current line; otherwise like `O`.
- p** Puts text after or below the cursor; otherwise like `P`.
- q** Unused.
- r** Replaces the single character at the cursor with a single character you type. The new character may be a RETURN; this is the easiest way to split lines. A count replaces each of the following count characters with the single character given; see `R` above which is the more usually useful iteration of `r`.

s	Changes the single character under the cursor to the text that follows up to an ESC; given a count, that many characters from the current line are changed. The last character to be changed is marked with \$ as in c.
t	Advances the cursor up to the character before the next character typed. Most useful with operators such as d and c to delete the characters up to a following character. You can use . to delete more if this doesn't delete enough the first time.
u	Undoes the last change made to the current buffer. If repeated, will alternate between these two states, thus is its own inverse. When used after an insert that inserted text on more than one line, the lines are saved in the numeric named buffers.
v	Unused.
w	Advances to the beginning of the next word, as defined by b.
x	Deletes the single character under the cursor. With a count deletes that many characters forward from the cursor position, but only on the current line.
y	An operator, yanks the following object into the unnamed temporary buffer. If preceded by a named buffer specification, "x, the text is placed in that buffer also. Text can be recovered by a later p or P.
z	Redraws the screen with the current line placed as specified by the following character: RETURN specifies the top of the screen, . the center of the screen, and '-' at the bottom of the screen. A count before the z gives the number of the line to place in the center of the screen instead of the default current line. To change the window size, use a count after the z and before the RETURN, as in z 5<CR>.
{	Retreats to the beginning of the preceding paragraph. A paragraph begins at each macro in the <i>paragraphs</i> option, normally .IP, .LP, .PP, .QP, and .bp. A paragraph also begins after a completely empty line, and at each section boundary (see [ [ above).
	Places the cursor on the character in the column specified by the count.
}	Advances to the beginning of the next paragraph. See { for the definition of paragraph.
~	Unused.
CTRL-C (DEL)	Interrupts the editor, returning it to command accepting state.

## 2.10. Terminal Information

vi works on a large number of display terminals. You can edit a terminal description file to drive new terminals. While it is advantageous to have an intelligent terminal that can locally insert and delete lines and characters from the display, vi functions quite well on dumb terminals over slow phone lines. vi allows for the low bandwidth in these situations and uses smaller window sizes and different display updating algorithms to make best use of the limited speed available.

You can also use the vi command set on hardcopy terminals, storage tubes and 'glass ttys' using a one-line editing window.

### Specifying Terminal Type

Before you can start vi you must tell the system what kind of terminal you are using. Here is a (necessarily incomplete) list of terminal type codes. If your terminal does not appear here, you should consult with one of the staff members on your system to find out the code for your terminal. If your terminal does not have a code, one can be assigned and a description for the terminal can be created.

Table 2-6 *Terminal Types*

Code	Full Name	Type
sun	Sun Workstation	Intelligent
tvi925	Televideo 925	Dumb
wy-50	Wyse 50	Dumb
2621	Hewlett-Packard 2621A/P	Intelligent
2645	Hewlett-Packard 264x	Intelligent
act4	Microterm ACT-IV	Dumb
act5	Microterm ACT-V	Dumb
adm3a	Lear Siegler ADM-3a	Dumb
adm31	Lear Siegler ADM-31	Intelligent
c100	Human Design Concept 100	Intelligent
dm1520	Datamedia 1520	Dumb
dm2500	Datamedia 2500	Intelligent
dm3025	Datamedia 3025	Intelligent
fox	Perkin-Elmer Fox	Dumb
h1500	Hazeltine 1500	Intelligent
h19	Heathkit h19	Intelligent
i100	Infoton 100	Intelligent
mime	Imitating a smart act4	Intelligent
t1061	Teleray 1061	Intelligent
vt52	Dec VT-52	Dumb

Suppose for example that you have a Hewlett-Packard HP2621A terminal. The code used by the system for this terminal is '2621'. In this case you can use one of the following commands to tell the system your terminal type:

```
hostname% setenv TERM 2621
```

If you are using the Bourne shell, use:

```
$ TERM=2621
$ export TERM
```

If you want to arrange to have your terminal type set up automatically when you log in, use the *tset* program. If you dial in on a *mime*, but often use hardwired ports, a typical line for your *.login* file (if you use *cs*) is

```
setenv TERM `tset - -d mime`
```

or for your *.profile* file (if you use *sh*):

```
TERM=`tset - -d mime`
```

*tset* knows which terminals are hardwired to each port and needs only to be told that when you dial in you are probably on a *mime*. You can use *tset* to change the erase and kill characters, too.

### Special Arrangements for Startup

*vi* takes the value of *\$TERM* and looks up the characteristics of that terminal in the file */etc/termcap*. If you don't know *vi*'s name for the terminal you are working on, look in */etc/termcap*. The editor adopts the convention that a null string in the environment is the same as not being set. This applies to *TERM*, *TERMCAP*, and *EXINIT*.

When *vi* starts, it attempts to read the variable *EXINIT* from your environment. If that exists, it takes the values in it as the default values for certain of its internal constants. If *EXINIT* doesn't exist, you will get all the normal defaults.

Should you inadvertently hang up the phone while inside *vi*, or should something else go wrong, all may not be lost. Upon returning to the system, type:

```
hostname% vi -r filename
```

This will normally recover the file. If there is more than one temporary file for a specific filename, *vi* recovers the newest one. You can get an older version by recovering the file more than once. The command *vi -r* without a filename lists the files from an on-line list that were saved in the last system crash (but *not* the file just saved when the phone was hung up).

### Open Mode on Hardcopy Terminals and 'Glass tty's'

If you are on a hardcopy terminal or a terminal that does not have a cursor that can move off the bottom line, you can still use the command set of *vi*, but in a different mode. When you give a *vi* command, the editor will tell you that it is using *open* mode. This name comes from the *open* command in *ex*, which is used to get into the same mode.

The only difference between *visual* mode and *open* mode is the way the text is displayed. In *open* mode the editor uses a single-line window into the file, and moving backward and forward in the file displays new lines, always below the current line. Two *vi* commands that work differently in *open* mode are:

- *z* and
- CTRL-R.

The *z* command does not take parameters, but rather draws a window of context around the current line and then returns you to the current line.

If you are on a hardcopy terminal, the CTRL-R command retypes the current line. On such terminals, vi normally uses two lines to represent the current line. The first line is a copy of the line as you started to edit it, and you work on the line below this line. When you delete characters, the editor types a number of \s to show you the characters that are deleted. It also reprints the current line soon after such changes so that you can see what the line looks like again.

It is sometimes useful to use this mode on very slow terminals that can support vi in the full screen mode. You can do this by entering `ex` and using an `open` command.

## Editing on Slow Terminals

When you are on a slow terminal, it is important to limit the amount of output that is generated to your screen so that you will not suffer long delays, waiting for the screen to be refreshed. We have already pointed out how the editor optimizes the updating of the screen during insertions on dumb terminals to limit the delays, and how the editor erases lines to @ when they are deleted on dumb terminals.

The use of the slow terminal insertion mode is controlled by the *slowopen* option. You can force the editor to use this mode even on faster terminals by giving the command:

```
:se slow<CR>
```

If your system is sluggish this helps lessen the amount of output coming to your terminal. You can disable this option by:

```
:se noslow<CR>
```

The editor can simulate an intelligent terminal on a dumb one. Try giving the command:

```
:se redraw<CR>
```

This simulation generates a great deal of output and is generally tolerable only on lightly loaded systems and fast terminals. You can disable this by giving the command:

```
:se noredraw<CR>
```

The editor also makes editing more pleasant at low speed by starting editing in a small window, and letting the window expand as you edit. This works particularly well on intelligent terminals. The editor can expand the window easily when you insert in the middle of the screen on these terminals. If possible, try the editor on an intelligent terminal to see how this works.

You can control the size of the window that is redrawn each time the screen is cleared by giving window size as an argument to the commands that cause large screen motions:

```
: / ? [[ ] ` ^
```

Thus if you are searching for a particular instance of a common string in a file, you can precede the first search command by a small number, say 3, and the editor will draw three line windows around each instance of the string it locates.

You can expand or contract the window size, placing the current line as you choose, with the `z` command, as in `z5<CR>`, which changes the window to five lines. You can also use `.` or `-`. Thus the command `z5.` redraws the screen with the current line in the center of a five-line window. Note that the command `5z.` has an entirely different effect, placing line 5 in the center of a new window. Use `-`, as in `5z-` to position the cursor at line 5 in the file.

The default window sizes are 8 lines at 300 baud, 16 lines at 1200 baud, and full-screen size at 9600 baud. Any baud rate less than 1200 behaves like 300, and any over 1200 like 9600.

If the editor is redrawing or otherwise updating large portions of the display, you can interrupt this updating by typing a DEL or RUB as usual. If you do this, you may partially confuse the editor about what is displayed on the screen. You can still edit the text on the screen if you wish; clear up the confusion by typing a CTRL-L, or you can move or search through the file again, ignoring the current state of the display.

See the section on *open* mode for another way to use the `vi` command set on slow terminals.

## Upper-case Only Terminals

If your terminal has only upper-case characters, you can still use `vi` by using the normal system convention for typing on such a terminal. Characters that you normally type are converted to lower case, and you can type upper-case letters by preceding them with a `'`. The characters `{ } | `` are not available on such terminals, but you can escape them as `\(\' \) ! ! \'`. These characters are represented on the display in the same way they are typed.<sup>9</sup>

## 2.11. Command Summary

The following is a quick summary of frequently used commands. Refer to the quick reference pages for a reference summary of all commands.

---

<sup>9</sup> The `'` character you give will not echo until you type another key.

Table 2-7 *Frequently-Used vi Commands*

Command	Description
SPACE	advance the cursor one column
CTRL-B	scroll backward one window
CTRL-D	scroll down in the file half a window
CTRL-E	exposes another line at the bottom of the window
CTRL-F	scroll forward one window
CTRL-G	tell what is going on
CTRL-H	backspace the cursor
CTRL-N	move down to next line, same column
CTRL-P	move up to previous line, same column
CTRL-U	scroll up in the file half a window
CTRL-Y	expose another line at the top of the window
+	move down to the next line, at the beginning of the line
-	move up to the previous line, at the beginning of the line
/	scan forward in the file for the following string
?	scan backward in the file for the following string
B	move the cursor back one word, ignoring punctuation
b	move the cursor back a word or punctuation character
E	move the cursor to the end of the current word ignoring punctuation
e	move the cursor to the end of the current word
G	go to specified line; default is last line in file
H	move the cursor to the top (or head) of the window
L	move the cursor to the last screen line
M	move the cursor to the middle screen line
n	scan through file for next instance of / or ? pattern
W	move the cursor forward one word, ignoring punctuation
w	move the cursor forward a word or punctuation character
CTRL-W	erase a word during an insert
DEL	your erase character (or CTRL-H), erases a character during an insert
CTRL-U	your kill character (or CTRL-X), kills the insert on this line
.	repeats the changing command
A	appends text at the end of the current line
a	appends text after the cursor
C	changes entire line
c	changes the object you specify to the following text
D	deletes to the end of a line
d	deletes the object you specify
I	inserts text at the beginning of a line
i	inserts text before the cursor
O	opens and inputs new lines, above the current line
o	opens and inputs new lines, below the current line
U	undoes the changes you made to the current line
u	undoes the last change

<b>^</b>	move cursor to first non-blank on line
<b>\$</b>	move cursor to end of line
<b>)</b>	move cursor forward one sentence
<b>(</b>	move cursor backward one sentence
<b>}</b>	move cursor forward one paragraph
<b>{</b>	move cursor backward one paragraph
<b>]]</b>	move cursor forward one section
<b>[[</b>	move cursor backward one section
<b>Fx</b>	find <i>x</i> backward in line
<b>fx</b>	find <i>x</i> forward in line
<b>P</b>	put text back, before cursor or above current line
<b>p</b>	put text back, after cursor or below current line
<b>Y</b>	yank one line into buffer
<b>y</b>	yank the object you specify into buffer; for copies and moves
<b>tx</b>	perform some operation forward on the line to <i>x</i>
<b>Tx</b>	perform some operation backward on the line to <i>x</i>

## Vi Quick Reference

### Entering and Leaving vi

% vi <i>name</i>	edit <i>name</i> at top
% vi + <i>n name</i>	... at line <i>n</i>
% vi + <i>name</i>	... at end
% vi -r	list saved files
% vi -r <i>name</i>	recover file <i>name</i>
% vi <i>name</i> ...	edit first; rest via :n
% vi -t <i>tag</i>	start at <i>tag</i>
% vi +/ <i>pat name</i>	search for <i>pat</i>
% view <i>name</i>	read only mode
ZZ	exit from vi, saving changes
CTRL-Z	stop vi for later resumption

### The Display

Last line	Error messages, echoing input to : / ? and !, feedback about i/o and large changes.
@ lines	On screen only, not in file.
~ lines	Lines past end of file.
CTRL-x	Control characters, DEL is delete.
tabs	Expand to spaces, cursor at last.

### Vi Modes

Command	Normal and initial state. Others return here. ESC (escape) cancels partial command.
Insert	Entered by a i A I o O c C s S R. Arbitrary text then terminates with ESC character, or abnormally with interrupt.
Last line	Reading input for : / ? or !; terminate with ESC or CR to execute, interrupt to cancel.

### Counts Before vi Commands

line/column number	z G
scroll amount	CTRL-D CTRL-U
replicate insert	a i A I
repeat effect	most rest

### Simple Commands

dw	delete a word
de	... leaving punctuation
dd	delete a line
3dd	... 3 lines
itextESC	insert text <i>abc</i>
cwnewESC	change word to <i>new</i>
easESC	pluralize word
xp	transpose characters

### Interrupting, Cancelling

ESC	end insert or incomplete cmd
CTRL-C	interrupt (or DEL)
CTRL-L	refresh screen if scrambled

### File Manipulation

:w	write back changes
:wq	write and quit
:q	quit
:q!	quit, discard changes
:e <i>name</i>	edit file <i>name</i>
:e!	reedit, discard changes
:e + <i>name</i>	edit, starting at end
:e + <i>n</i>	edit starting at line <i>n</i>
:e #	edit alternate file
CTRL-^	synonym for :e #
:w <i>name</i>	write file <i>name</i>
:w! <i>name</i>	overwrite file <i>name</i>
:sh	run shell, then return
!: <i>cmd</i>	run <i>cmd</i> , then return
:n	edit next file in arglist
:n <i>args</i>	specify new arglist
:f	show current file and line
CTRL-G	synonym for :f
:ta <i>tag</i>	to tag file entry <i>tag</i>
CTRL-]	:ta, following word is <i>tag</i>

### Positioning within File

CTRL-F	forward screenfull
CTRL-B	backward screenfull
CTRL-D	scroll down half screen
CTRL-U	scroll up half screen
G	goto line (end default)
/ <i>pat</i>	next line matching <i>pat</i>
? <i>pat</i>	prev line matching <i>pat</i>
n	repeat last / or ?
N	reverse last / or ?
/ <i>pat</i> + <i>n</i>	n'th line after <i>pat</i>
? <i>pat</i> ?- <i>n</i>	n'th line before <i>pat</i>
]]	next section/function
[[	previous section/function
%	find matching ( ) { or }

### Adjusting the Screen

CTRL-L	clear and redraw
CTRL-R	retype, eliminate @ lines
zCR	redraw, current at window top
z-	... at bottom
z.	... at center
/ <i>pat</i> /z- <i>n</i> .	<i>pat</i> line at bottom
zn.	use <i>n</i> line window
CTRL-E	scroll window down 1 line
CTRL-Y	scroll window up 1 line

**Marking and Returning**

``	previous context
``	... at first non-white in line
mx	mark position with letter <i>x</i>
`x	to mark <i>x</i>
ˆx	... at first non-white in line

**Line Positioning**

H	home window line
L	last window line
M	middle window line
+	next line, at first non-white
-	previous line, at first non-white
CR	return, same as +
↓ or j	next line, same column
↑ or k	previous line, same column

**Character Positioning**

^	first non-blank
0	beginning of line
\$	end of line
h or →	forward
l or ←	backwards
CTRL-H	same as ←
space	same as →
fx	find <i>x</i> forward
Fx	<i>f</i> backward
tx	upto <i>x</i> forward
Tx	back upto <i>x</i>
;	repeat last <i>f F t</i> or <i>T</i>
,	inverse of ;
	to specified column
%	find matching ( { ) or }

**Words, Sentences, Paragraphs**

w	word forward
b	back word
e	end of word
)	to next sentence
}	to next paragraph
(	back sentence
{	back paragraph
W	blank delimited word
B	back W
E	to end of W

**Commands for LISP**

)	Forward s-expression
}	... but don't stop at atoms
(	Back s-expression
{	... but don't stop at atoms

**Corrections During Insert**

CTRL-H	erase last character
CTRL-W	erases last word
erase	your erase, same as CTRL-H
kill	your kill, erase input this line
\	escapes CTRL-H, your erase and kill
ESC	ends insertion, back to command
CTRL-C	interrupt, terminates insert
CTRL-D	backtab over <i>autoindent</i>
CTRL-^D	kill <i>autoindent</i> , save for next
0CTRL-D	... but at margin next also
CTRL-V	quote non-printing character

**Insert and Replace**

a	append after cursor
i	insert before
A	append at end of line
I	insert before first non-blank
o	open line below
O	open above
rx	replace single char with <i>x</i>
R	replace characters

**Operators (double to affect lines)**

d	delete
c	change
<	left shift
>	right shift
!	filter through command
=	indent for LISP
y	yank lines to buffer

**Miscellaneous Operations**

C	change rest of line
D	delete rest of line
s	substitute chars
S	substitute lines
J	join lines
x	delete characters
X	... before cursor
Y	yank lines

**Yank and Put**

p	put back lines
P	put before
"xp	put from buffer <i>x</i>
"xy	yank to buffer <i>x</i>
"xd	delete into buffer <i>x</i>

**Undo, Redo, Retrieve**

u	undo last change
U	restore current line
.	repeat last change
"dp	retrieve <i>d</i> 'th last delete

---

## Command Reference for the `ex` Line Editor

Command Reference for the <code>ex</code> Line Editor .....	71
3.1. Using <code>ex</code> .....	71
3.2. File Manipulation .....	72
Current File .....	72
Alternate File .....	72
Filename Expansion .....	72
3.3. Special Characters .....	73
Multiple Files and Named Buffers .....	73
Read-Only Mode .....	73
3.4. Exceptional Conditions .....	73
Errors and Interrupts .....	73
Recovering If Something Goes Wrong .....	73
3.5. Editing Modes .....	74
3.6. Command Structure .....	74
Specifying Command Parameters .....	74
Invoking Command Variants .....	75
Flags After Commands .....	75
Writing Comments .....	75
Putting Multiple Commands on a Line .....	75
Reporting Large Changes .....	75
3.7. Command Addressing .....	75
Addressing Primitives .....	75
Combining Addressing Primitives .....	76

3.8. Regular Expressions and Substitute Replacement Patterns .....	76
Regular Expressions .....	76
Magic and Nomagic .....	76
Basic Regular Expression Summary .....	77
Combining Regular Expression Primitives .....	77
Substitute Replacement Patterns .....	78
3.9. Command Reference .....	78
3.10. Option Descriptions .....	88
3.11. Limitations .....	94

---

## Command Reference for the `ex` Line Editor

This chapter<sup>10</sup> provides reference material for `ex`, the line-oriented text editor, which also supports display oriented editing in the form of the `vi` editor described in the chapter "Using `vi`, the Visual Display Editor". The contents of this chapter describe the line-oriented part of `ex`. You can also use these commands with `vi`. For a summary of `ex` commands, see the `ex` Quick Reference.

### 3.1. Using `ex`

`ex` has a set of options, which you can use to tailor `ex` 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. To simplify the description which follows, we assume the default settings of the options, and we assume that you are running `ex` on a Sun Workstation.

If there is a variable `EXINIT` in the environment, `ex` executes the commands in that variable, otherwise if there is a file `.exrc` in your HOME directory `ex` reads commands from that file, simulating a *source* command. Option setting commands placed in `EXINIT` or `.exrc` are executed before each editor session.

If you are running `ex` on a terminal, `ex` determines the terminal type from the `TERM` variable in the environment when invoked. If there is a `TERMCAP` variable in the environment, and the type of the terminal described there matches the `TERM` variable, that description is used. Also if the `TERMCAP` variable contains a pathname (beginning with a `/`), `ex` seeks the description of the terminal in that file, rather than in the default `/etc/termcap`.)

The standard `ex` command format follows. Brackets '[' ']' surround optional parameters here.

```
hostname% ex [ - ] [-v] [-t tag] [-r] [-l] [-wn] [-x] [-R]
[+command ] filename ...
```

The most common case edits a single file with no options, that is,:

```
hostname% ex filename
```

The `'-'` command line option option suppresses all interactive-user feedback and is useful in processing `ex` scripts in command files. The `-v` option is

---

<sup>10</sup> The material in this chapter is derived from *Ex Reference Manual*, W.N. Joy, M. Horton, University of California, Berkeley.

equivalent to using `vi` rather than `ex`. The `-t` option is equivalent to an initial `tag` command, editing the file containing the `tag` and positioning the editor at its definition.

Use the `-r` option to recover a file after an editor or system problem, retrieving the last saved version of the named file or, if no file is specified, displaying a list of saved files. The `-l` option sets up for editing LISP, setting the `showmatch` and `lisp` options. The `-w` option sets the default window size to `n`, and is useful on dialups to start in small windows. The `-x` option causes `ex` to prompt for a `key`, which is used to encrypt and decrypt the contents of the file, which should already be encrypted using the same key (see `crypt` in the *Commands Reference Manual for the Sun Workstation*). The `-R` option sets the `readonly` option at the start. If set, writes will fail unless you use an `!` after the write. This option affects `ZZ`, `autowrite` and anything that writes to guarantee you won't clobber a file by accident. *Filename* arguments indicate files to be edited. An argument of the form `+command` indicates that the editor should begin by executing the specified command. If `command` is omitted, it defaults to '\$', initially positioning `ex` at the last line of the first file. Other useful commands here are scanning patterns of the form `'/pat'` or line numbers, such as `+100`, which means 'start at line 100.'

## 3.2. File Manipulation

The following describes commands for handling files.

### Current File

`ex` normally edits the contents of a single file, whose name is recorded in the *current* filename. `ex` performs all editing actions in a buffer 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 you write the buffer contents 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. When a file is edited, its name becomes the current filename, and its contents are read into the buffer.

The current file is almost always considered to be *edited*. This means that the contents of the buffer are logically connected with the current filename, 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*, `ex` will not normally write on it if it already exists. The `file` command will say [Not edited] if the current file is not considered edited.

### Alternate File

Each time a new value is given to the current filename, the previous current filename is saved as the *alternate* filename. Similarly if a file is mentioned but does not become the current file, it is saved as the alternate filename.

### Filename Expansion

You may specify filenames within the editor using the normal Shell expansion conventions. In addition, the character `%` in filenames is replaced by the *current* filename and the character `#` by the *alternate* filename. 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.

### 3.3. Special Characters

Some characters take on special meanings when used in context searches and in patterns given to the *substitute* command. For *edit*, these are the caret (^) and dollar sign (\$) characters, meaning the beginning and end of a line, respectively. *ex* has the following additional special characters:

. & \* [ ] ~

To use one of the special characters as its simple graphic representation rather than with its special meaning, precede it by a backslash (\). The backslash always has a special meaning.

### Multiple Files and Named Buffers

If more than one file is given on the *ex* command line, the first file is edited as described above. The remaining arguments are placed with the first file in the *argument list*. You can display the current argument list with the *args* command. To edit the next file in the argument list, use the *next* command. You may also respecify the argument list 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.

To save blocks of text while editing, and especially when editing more than one file, *ex* has a group of named buffers. These are similar to the normal buffer, except that only a limited number of operations are available on them. The buffers have names *a* through *z*. It is also possible to refer to *A* through *Z*; the upper-case buffers are the same as the lower but commands append to named buffers rather than replacing if upper-case names are used.

### Read-Only Mode

It is possible to use *ex* in *read only* mode to look at files that you have no intention of modifying. This mode protects you from accidentally overwriting the file. Read only mode is on when the *readonly* option is set. It can be turned on with the *-R* command line option, by the *view* command line invocation, or by setting the *readonly* option. It can be cleared by setting *noreadonly*. It is possible to write, even while in read only mode, by indicating that you really know what you are doing. You can write to a different file with *:w newfilename*, or can use the *:w!* form of write, even while in read only mode.

### 3.4. Exceptional Conditions

The following describes additional editing situations.

#### Errors and Interrupts

When errors occur *ex* flashes the workstation screen and displays an error diagnostic. If the primary input is from a file, editor processing terminates. If you interrupt *ex*, it displays 'Interrupt' and returns to its command level. If the primary input is a file, *ex* exits when this occurs.

#### Recovering If Something Goes Wrong

If something goes wrong and the buffer has been modified since it was last written out, or if the system crashes, either the editor or the system (after it reboots) attempts to preserve the buffer. The next time you log in, you should be able to recover the work you were doing, losing at most a few lines of changes from the last point before the problem. To recover a file, use the *-r* option. If you were editing the file *resume* for example, change to the directory where you were when the problem occurred, and use *ex* with the *-r* (recover) option:

```
hostname% ex -r file
```

After checking that the retrieved file is indeed ok, you can *write* it over the previous contents of that file.

You will normally get mail from the system telling you when a file has been saved after the system has gone down. Use the `-r` option without a following filename:

```
hostname% ex -r
```

to display a list of the files that have been saved for you. In the case of a hangup, the file will not appear in the list, although it can be recovered.

### 3.5. Editing Modes

`ex` has five distinct modes. The primary mode is *command* mode. You type in commands in command mode when a `:` prompt is present, and execute them each time you send a complete line. In *insert* mode, `ex` gathers input lines and places them in the file. The *append*, *insert*, and *change* commands use insert mode. No prompt is displayed when you are in text input mode. To leave this mode and return to command mode, type a `.` alone at the beginning of a line.

The last three modes are *open* and *visual* modes, entered by the commands of the same names, and, within open and visual modes *text insertion* mode. In *open* and *visual* modes, you do local editing operations on the text in the file. The *open* command displays one line at a time on the screen, while *visual* works on the workstation and CRT terminals with random positioning cursors, using the screen as a single window for file editing changes. See the chapter on "Using `vi`, The Visual Display Editor" for descriptions of these modes.

### 3.6. Command Structure

Most command names are English words; you can use initial prefixes of the words as acceptable abbreviations. The ambiguity of abbreviations is resolved in favor of the more commonly used commands. As an example, the command `substitute` can be abbreviated as `s` while the shortest available abbreviation for the `set` command is `se`. See the "Command Reference" section for descriptions and acceptable abbreviations.

#### Specifying Command Parameters

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. Counts are rounded down if necessary. Thus the command `10p` displays the tenth line in the buffer, while `d5` deletes five lines from the buffer, starting with the current line.

Some commands take other information or parameters, that you provide after the command name. Examples would be option names in a `set` command such as, `set number`, a filename in an `edit` command, a regular expression in a `substitute` command, or a target address for a `copy` command, such as, `1,5 copy 25`.

**Invoking Command Variants** A number of commands have two distinct variants. The variant form of the command is invoked by placing an `!` immediately after the command name. You can control some of the default variants with options; in this case, the `!` serves to toggle the default.

**Flags After Commands** You may place the characters `#`, `p` and `l` after many commands. You must precede a `p` or `l` by a blank or tab except in the single special case of `dp`. The command that these characters abbreviates is executed after the command completes. Since `ex` normally shows the new current line after each change, `p` is rarely necessary. You can also give any number of `+` or `-` characters with these flags. If they appear, the specified offset is applied to the current line value before the display command is executed.

**Writing Comments** It is possible to give editor commands which are ignored. This is useful when making complex editor scripts for which comments are desired. Use the double quote `"` as the comment character. Any command line beginning with `"` is ignored. You can also put comments beginning with `"` at the ends of commands, except in cases where they could be confused as part of text, for example as shell escapes and the `substitute` and `map` commands.

**Putting Multiple Commands on a Line** You can place more than one `ex` command on a line by separating each pair of commands by a pipe (`|`) character. However the *global* commands, comments, and the shell escape `!` must be the last command on a line, as they are not terminated by a `|`.

**Reporting Large Changes** Most commands which change the contents of the editor buffer give feedback if the scope of the change exceeds a threshold given by the *report* option. This feedback helps to detect undesirably large changes so that you may quickly and easily reverse them with *undo*. 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.

**3.7. Command Addressing** The following describes the editor commands called *addressing primitives*.

#### Addressing Primitives

- `.` The current line. The current line is traditionally called 'dot' because you address it with a dot `'.'`. Most commands leave the current line as the last line which they affect. The default address for most commands is the current line, so you rarely use `'.'` alone as an address.
- `n` The *n*th line in the editor's buffer, lines being numbered sequentially from 1.
- `$` The last line in the buffer.
- `%` An abbreviation for `1, $`, the entire buffer.
- `+n -n` An offset relative to the current buffer line. The forms `+.3 +3` and `+++` are all equivalent; if the current line is line 100, they all address line 103.

`/pat/ ?pat?`

Scan forward and backward respectively for a line containing *pat*, a regular expression (as defined below in the section "Regular Expressions and Substitute Replacement Patterns". The scans normally wrap around the end of the buffer. If all that is desired is to show the next line containing *pat*, you may omit trailing / or ?. If you omit *pat* or leave it explicitly empty, the last regular expression specified is located. The forms `\` and `\?` scan using the last regular expression used in a scan; after a `substitute`, `//` and `??` would scan using the substitute's regular expression.

``` `x`

Before each non-relative motion of the current line '.', the previous current line is marked with a tag, subsequently referred to as '``'. This makes it easy to refer or return to this previous context. You can also establish marks with the `mark` command, using single lower-case letters *x* and the marked lines referred to as '*x*'.

### Combining Addressing Primitives

Addresses to commands consist of a series of addressing primitives, 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 you give more addresses than the command requires, 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 '.'. So ',100' is equivalent to ',.100'. It is an error to give a prefix address to a command which expects none.

## 3.8. Regular Expressions and Substitute Replacement Patterns

### Regular Expressions

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. `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 regular expression, that is `//` 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` and `vi` 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 backslash (\) to use them as "ordinary" characters. With *nomagic*, the default for `edit`, regular expressions are much simpler because there are only two metacharacters: '^' (beginning of line) and '\$' (end of line). 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 the setting of this option is *magic*<sup>11</sup>

## Basic Regular Expression Summary

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

<i>char</i>	An ordinary character matches itself. The characters <code>^</code> at the beginning of a line, <code>\$</code> at the end of line, <code>*</code> as any character other than the first, <code>'</code> , <code>\</code> , <code>[</code> , and <code>~</code> are not ordinary characters and must be escaped (preceded) by <code>\</code> to be treated as such.
<code>^</code>	At the beginning of a pattern forces the match to succeed only at the beginning of a line.
<code>\$</code>	At the end of a regular expression forces the match to succeed only at the end of the line.
<code>.</code>	Matches any single character except the new-line character.
<code>\&lt;</code>	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.
<code>\&gt;</code>	Similar to <code>\&lt;</code> , but matching the end of a 'variable' or 'word,' that is either the end of the line or before character which is neither a letter, nor a digit, nor the underline character.
<code>[string]</code>	Matches any single character in the class defined by <i>string</i> . Most characters in <i>string</i> define themselves. A pair of characters separated by <code>-</code> in <i>string</i> defines a set of characters between the specified lower and upper bounds, thus <code>[a-z]</code> as a regular expression matches any single lower-case letter. If the first character of <i>string</i> is a <code>^</code> , the construct matches all but those characters; thus <code>[^a-z]</code> matches anything but a lower-case letter and of course a newline. You must escape any of the characters <code>^</code> , <code>[</code> , or <code>-</code> in <i>string</i> with a preceding <code>\</code> .

## Combining Regular Expression Primitives

The concatenation of two regular expressions matches the leftmost and then longest string, which can be divided with the first piece matching the first regular expression and the second piece matching the second. 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

<sup>11</sup> To discern what is true with *nomagic* it is sufficient 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 `\`. With *nomagic* the characters `~` and `&` also lose their special meanings related to the replacement pattern of a `substitute`.

effects in the *substitute* replacement patterns.

## Substitute Replacement Patterns

The basic metacharacters for the replacement pattern are `&` and `~`; these are given as `\&` and `\~` when *nomagic* is set. Each instance of `&` is replaced by the characters which the regular expression matched. The metacharacter `'~'` stands, in the replacement pattern, for the defining text of the previous replacement pattern.

Other metasequences possible in the replacement pattern are always introduced by the escape character `\`. The sequence `'\n'` is replaced by the text matched by the *n*-th regular subexpression enclosed between `\(` and `\)`.<sup>12</sup> 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.

## 3.9. Command Reference

The following form is a prototype for all `ex` commands:

*address command ! parameters count flags*

All parts are optional; the simplest case is the empty command, which displays the next line in the file. To avoid confusion from within *visual* mode, `ex` ignores a `:` preceding any command.

In the following command descriptions, the default addresses are shown in parentheses, which are *not*, however, part of the command.

`abbreviate word rhs`                    abbr: ab

Add the named abbreviation to the current list. When in input mode in *visual*, if *word* is typed as a complete word, it will be changed to *rhs*.

`( . ) append`                            abbr: a  
*text*

.

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.

a!  
*text*

.

The variant flag to `append` toggles the setting for the *autoindent* option during the input of *text*.

<sup>12</sup> When nested, parenthesized subexpressions are present, *n* is determined by counting occurrences of `\(` starting from the left.

args

The members of the argument list are printed, with the current argument delimited by [ and ].

( . , . ) change *count*           abbr: c  
*text*

.

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`.

c!  
*text*

.

The variant toggles *autoindent* during the change.

( . , . ) copy *addr flags*           abbr: co

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

( . , . ) delete *buffer count flags*abbr: d

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. If a named *buffer* is specified by giving a letter, then the specified lines are saved in that buffer, or appended to it if an upper case letter is used.

edit *file*                           abbr: e  
ex *file*  
edit! *file*

Used to begin an editing session on a new file. Same as `:vi file`. The editor 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 command is aborted. The command otherwise deletes the entire contents of the editor buffer, makes the named file the current file and prints the new filename. After insuring that this file is sensible the editor reads the file into its buffer. A 'sensible' file 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.

If the read of the file completes without error, the number of lines and characters read is typed. If there were any 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



`g! /pat/ cmds`                   abbr: v

The variant form of `global` runs `cmds` at each line not matching `pat`.

`(.)insert`                       abbr: i  
*text*

.

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.

`i!`  
*text*

.

The variant toggles `autoindent` during the `insert`.

`(.,.+1) join count flags`   abbr: j

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, two if there was a `'.'` at the end of the line, or none if the first following character is a `)`. 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.

`j!`

The variant causes a simpler `join` with no white space processing; the characters in the lines are simply concatenated.

`(.) k x`

The `k` command is a synonym for `mark`. It does not require a blank or tab before the following letter.

`(.,.) list count flags`

Prints the specified lines in a more unambiguous way: tabs are printed as CTRL-I (^I) and the end of each line is marked with a trailing `$`. The current line is left at the last line printed.

`map lhs rhs`

The `map` command is used to define macros for use in *visual* mode. *lhs* should be a single character, or the sequence `#n`, for *n* a digit, referring to function key *n*. When this character or function key is typed in *visual* mode, it will be as though the corresponding *rhs* had been typed. On terminals without function keys, you can type `#n`. See the "Macros" section in the chapter "Using `vi`, the Visual Display Editor" for more details.

( . ) mark *x*

Gives the specified line mark *x*, a single lower case letter. The *x* must be preceded by a blank or a tab. The addressing form `'x` then addresses this line. The current line is not affected by this command.

( . , . ) move *addr*                   abbr: m

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

next                                   abbr: n

The next file from the command line argument list is edited.

n!

The variant suppresses warnings about the modifications to the buffer not having been written out, discarding (irretrievably) any changes that may have been made.

n *filelist*

n +*command 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. If *command* is given (it must contain no spaces), then it is executed after editing the first such file.

( . , . ) number *count flags*   abbr: # or nu

Prints each specified line preceded by its buffer line number. The current line is left at the last line printed. The *count* option specifies the number of lines to print.

( . ) open *flags*                   abbr: o

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

Enters intraline editing *open* 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. To exit this mode, use `Q`. See the chapter on "Using vi the Visual Display Editor".

preserve

The current editor buffer is saved as though the system 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.

( . , . )print *count*                   abbr: p or P

Prints the specified lines with non-printing characters printed as control characters '^X'; delete (hexadecimal 0x7f) is represented as ^?. The *count* option specifies the number of lines to print. The current line is left at the last line printed.

( . )put *buffer*                       abbr: pu

Puts back previously deleted or yanked lines. Normally used with `delete` to effect movement of lines, or with `yank` to effect duplication of lines. If no *buffer* is specified, then the last deleted or yanked text is restored. But no modifying commands may intervene between the `delete` or `yank` and the `put`, nor may lines be moved between files without using a named buffer. By using a named buffer, text may be restored that was saved there at any previous time.

quit                                   abbr: q

Causes `ex` to terminate. 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 does not *quit*. `ex` also warns you if there are more files in the argument list. Normally, you do want to save your changes, so you should use a `write` command; if you wish to discard them, use the `q!` command variant.

q!

Quits from the editor, discarding changes to the buffer without complaint.

( . ) read *file*                       abbr: r

Places a copy of the text of the given file in the editing buffer after the specified line. If no *file* is given the current file name is used. The current file name is not changed unless there is none in which case *file* 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' (zero) 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. Within *open* and *visual* modes the current line is set to the first line read rather than the last.

( . ) read !*command*

Reads the output of the command *command* into the buffer after the specified line. This is not a variant form of the command, rather a `read` specifying a *command* rather than a *filename*; a blank or tab before the `!` is mandatory.



stop

Suspends the editor, returning control to the top level shell. If *autowrite* is set and there are unsaved changes, a write is done first unless the form *stop!* is used. This command is only available where supported by the teletype driver and operating system.

( . , . ) substitute *options count flags*abbr: s

If *pat* and *repl* are omitted, then the last substitution is repeated. This is a synonym for the & command.

( . , . ) t *addr flags*

The t command is a synonym for copy.

ta *tag*

The focus of editing switches to the location of *tag*, switching to a different line in the current file where it is defined, or if necessary to another file. If you have modified the current file before giving a *tag* command, you must write it out; giving another *tag* command, specifying no tag reuses the previous tag.

The tags file is normally created by a program such as *ctags*, and consists of a number of lines with three fields separated by blanks or tabs. The first field gives the name of the tag, the second the name of the file where the tag resides, and the third gives an addressing form which can be used by the editor to find the tag; this field is usually a contextual scan using */pat/* to be immune to minor changes in the file. Such scans are always performed as if *nomagic* were set.

The tag names in the tags file must be sorted alphabetically.

unabbreviate *word*                   abbr: una

Delete *word* from the list of abbreviations.

undo                                   abbr: u

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* commands.) Also, the commands *write* and *edit* which interact with the file system cannot be undone. *undo* is its own inverse.

*undo* always marks the previous value of the current line '.' as ' '. 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 its pre-command value after an *undo*.

unmap *lhs*

The macro expansion associated by `map` for *lhs* is removed.

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

A synonym for the global command variant `g!`, running the specified *cmds* on each line that does not match *pat*.

version abbr: ve

Prints the current version number of the editor as well as the date the editor was last changed.

vi *file*

Same as `:edit file` or `:ex file`.

( . ) visual *type count flags* abbr: vi

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. A *count* specifies an initial window size; the default is the value of the option *window*. See the chapter "Using vi, the Visual Display Editor for more details. To exit visual mode, type `Q`.

visual *file*

visual +*n* *file*

From visual mode, this command is the same as `edit`.

( 1, \$ ) write *file* abbr: w

Writes changes made back to *file*, printing the number of lines and characters written. Normally *file* is omitted and the text goes back where it came from. If a *file* is specified, then text will be written to that file.<sup>13</sup> 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 never changed.

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

( 1, \$ ) write>> *file* abbr: w>>

Writes the buffer contents at the end of an existing file.

---

<sup>13</sup> The editor writes 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`, `/dev/null`. Otherwise, you must give the variant form `w!` to force the write.







**directory, dir** default: dir=/tmp

Specifies the directory in which `ex` places its buffer file. If this directory is not writeable, then the editor will exit abruptly when it fails to be able to create its buffer there. This feature is useful on systems where `/tmp` fills up. Being able to specify that the editor use your own file space can allow you to edit even if `/tmp` is full.

**edcompatible** default: noedcompatible

Causes the presence or absence of `g` and `c` suffixes on substitute commands to be remembered, and to be toggled by repeating the suffixes. The suffix `r` makes the substitution be as in the `~` command, instead of like `&`.

**errorbells, eb** default: noeb

Error messages are preceded by a beep or bell.<sup>14</sup> If possible the editor always places the error message in a standout mode of the terminal (such as inverse video) instead of ringing the bell.

**hardtabs, ht** default: ht=8

Gives the boundaries on which terminal hardware tabs are set (or on which the system expands tabs).

**ignorecase, ic** default: noic

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.

**lisp** default: nolisp

*autoindent* indents appropriately for LISP code, and the `(, ), {, }, [[, and ]]` commands in *open* and *visual* modes are modified to have meaning for LISP.

**list** default: nolist

All printed lines will be displayed (more) unambiguously, showing tabs and ends-of-lines as in the `list` command.

**magic** default: magic for `ex` and `vi`<sup>15</sup>

If *nomagic* is set, the number of regular expression metacharacters is greatly reduced, with only `^` and `$` having special 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

---

<sup>14</sup> Beeping and bell ringing in *open* and *visual* on errors is not suppressed by setting *noeb*.

<sup>15</sup> *nomagic* for `edit`.

set by preceding them with a backslash (\).

**mesg** default: mesg

Causes write permission to be turned off to the terminal while you are in visual mode, if *nomesg* is set.

**number, nu** default: nonumber

Causes 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*, 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.

**optimize, opt** default: optimize

Throughput of text is expedited by setting the terminal to not do automatic carriage returns when printing more than one (logical) line of output, greatly speeding output on terminals without addressable cursors when text with leading white space is printed.

**paragraphs, para** default: para=IPLPPPQPP Libp

Specifies the paragraphs for the { and } operations in *open* and *visual* modes. The pairs of characters in the option's value are the names of the macros which start paragraphs.

**prompt** default: prompt

Command mode input is prompted for with a :.

**readonly, ro** default: off

If set, writes will fail unless you use an ! after the write. Affects *x*, *ZZ*, *autowrite* and anything that writes to guarantee you won't clobber a file by accident. Abbreviate to *ro*.

**redraw** default: noredraw

The editor simulates (using great amounts of output), an intelligent terminal on a dumb terminal (e.g. during insertions in *visual* mode the characters to the right of the cursor position are refreshed as each input character is typed.) Useful only at very high speed.

**remap** default: remap

If on, macros are repeatedly tried until they are unchanged. For example, if *o* is mapped to *O*, and *O* is mapped to *I*, then if *remap* is set, *o* will map to *I*, but if *noremmap* is set, it will map to *O*. Can map *q* to *#* and *#1* to something else, and *q1* to something else. If off, can map CTRL-L to *l* and CTRL-R to CTRL-L without having CTRL-R map to *l*.

**report** default: report=5<sup>16</sup>

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.

**scroll** default: scroll=1/2 window

Determines the number of logical lines scrolled when an end-of-file is received from a terminal input in command mode, and the number of lines printed by a command mode *z* command (double the value of *scroll*).

**sections** default: sections=SHNHH HU

Specifies the section macros for the `[ [ and ] ]` operations in *open* and *visual* modes. The pairs of characters in the options's value are the names of the macros which start paragraphs.

**shell, sh** default: sh=/bin/sh

Gives the path name of the shell forked for the shell escape command `!`, and by the `shell` command. The default is taken from SHELL in the environment, if present.

**shiftwidth, sw** default: sw=8

Gives the width a software tab stop, used in reverse tabbing with CTRL-D when using *autoindent* to append text, and by the shift commands.

**showmatch, sm** default: nosm

In *open* and *visual* mode, when a `)` or `}` is typed, move the cursor to the matching `(` or `{` for one second if this matching character is on the screen. Extremely useful with LISP.

---

<sup>16</sup> 2 for edit.



**w300, w1200, w9600**

These are not true options but set *window* only if the speed is slow (300), medium (1200), or high (9600), respectively. They are suitable for an EXINIT and make it easy to change the 8/16/full screen rule. Can specify a 12-line window at 300 baud and a 23-line window at 1200 in your EXINIT with: `:set w300=12 w1200=23`. Synonymous with *window* but only at 300, 1200, and 9600 baud.

**wrapscan, ws**                      default: ws

Searches using the regular expressions in addressing will wrap around past the end of the file.

**wrapmargin, wm**                      default: wm=0

Defines a margin for automatic wrapover of text during input in *open* and *visual* modes. Any number other than 0 (zero) is a distance from the right edge of the area where wraps can take place. If you type past the margin, the entire word is rewritten on the next line. Behaves much like *fill/justify* mode in *nroff*. See the section "Using *vi*, the Visual Display Editor" for details.

**writeany, wa**                      default: nowa

Inhibit the checks normally made before *write* commands, allowing a write to any file which the system protection mechanism will allow.

**3.11. Limitations**

Editor limits that the user is likely to encounter are as follows: 1024 characters per line, 256 characters per global command list, 128 characters per file name, 128 characters in the previous inserted and deleted text in *open* or *visual* modes, 100 characters in a shell escape command, 63 characters in a string valued option, and 30 characters in a tag name, and a limit of 250,000 lines in the file is silently enforced.

The *visual* implementation limits the number of macros defined with *map* to 32, and the total number of characters in macros to be less than 512.

## Ex Quick Reference

### Entering/Leaving ex

% <i>ex name</i>	edit <i>name</i> , start at end
% <i>ex +n name</i>	... at line <i>n</i>
% <i>ex -t tag</i>	start at <i>tag</i>
% <i>ex -r</i>	list saved files
% <i>ex -r name</i>	recover file <i>name</i>
% <i>ex name ...</i>	edit first; rest via : <i>n</i>
% <i>ex -R name</i>	read only mode
: <i>x</i>	exit, saving changes
: <i>q!</i>	exit, discarding changes

### ex States

Command	Normal and initial state. Input prompted for by :. Your kill character cancels partial command.
Insert	Entered by a <i>i</i> and <i>c</i> . Arbitrary text then terminates with line having only . character on it or abnormally with interrupt.
Open/visual	Entered by <i>open</i> or <i>vi</i> , terminates with <i>Q</i> or <i>^\</i> .

### ex Commands

abbrev	<b>ab</b>	next	<b>n</b>	unabbrev	<b>una</b>
append	<b>a</b>	number	<b>nu</b>	undo	<b>u</b>
args	<b>ar</b>	open	<b>o</b>	unmap	<b>unm</b>
change	<b>c</b>	preserve	<b>pre</b>	version	<b>ve</b>
copy	<b>co</b>	print	<b>p</b>	visual	<b>vi</b>
delete	<b>d</b>	put	<b>pu</b>	write	<b>w</b>
edit	<b>e</b>	quit	<b>q</b>	xit	<b>x</b>
file	<b>f</b>	read	<b>re</b>	yank	<b>ya</b>
global	<b>g</b>	recover	<b>rec</b>	window	<b>z</b>
insert	<b>i</b>	rewind	<b>rew</b>	escape	<b>!</b>
join	<b>j</b>	set	<b>se</b>	shift	<b>&lt;</b>
list	<b>l</b>	shell	<b>sh</b>	print next	<b>CR</b>
map		source	<b>so</b>	resubst	<b>&amp;</b>
mark	<b>ma</b>	stop	<b>st</b>	rshift	<b>&gt;</b>
move	<b>m</b>	substitute	<b>s</b>	scroll	<b>^D</b>

### ex Command Addresses

<i>n</i>	line <i>n</i>	<i>/pat</i>	next with <i>pat</i>
.	current	<i>?pat</i>	previous with <i>pat</i>
\$	last	<i>x-n</i>	<i>n</i> before <i>x</i>
+	next	<i>x,y</i>	<i>x</i> through <i>y</i>
-	previous	<i>^x</i>	marked with <i>x</i>
+ <i>n</i>	<i>n</i> forward	<i>^^</i>	previous context
%	1,\$		

### Specifying Terminal Type

% <i>setenv TERM type</i>	(for <i>cs</i> )
\$ <i>TERM=type; export TERM</i>	(for <i>sh</i> )

See also *tset* in the user's manual.

### Some Terminal Types

2621	43	adm31	dw1	h19
2645	733	adm3a	dw2	i100
300s	745	c100	gt40	mime
33	act4	dm1520	gt42	owl
37	act5	dm2500	h1500	t1061
4014	adm3	dm3025	h1510	vt52

### Initializing Options

EXINIT	place <i>set</i> 's here in environment var.
<i>set x</i>	enable option
<i>set nox</i>	disable option
<i>set x=val</i>	give value <i>val</i>
<i>set</i>	show changed options
<i>set all</i>	show all options
<i>set x?</i>	show value of option <i>x</i>

### Useful Options

autoindent	<b>ai</b>	supply indent
autowrite	<b>aw</b>	write before changing files
ignorecase	<b>ic</b>	in scanning ( ) { } are s-exp's
llsp		print ^I for tab, \$ at end
magic		. [ * special in patterns
number	<b>nu</b>	number lines
paragraphs	<b>para</b>	macro names which start ...
redraw		simulate smart terminal
scroll		command mode lines
sections	<b>sect</b>	macro names ...
shiftwidth	<b>sw</b>	for < >, and input ^D
showmatch	<b>sm</b>	to ) and } as typed
slowopen	<b>slow</b>	choke updates during insert
window		visual mode lines
wrapscan	<b>ws</b>	around end of buffer
wrapmargin	<b>wm</b>	automatic line splitting

### Scanning Pattern Formation

^	beginning of line
\$	end of line
.	any character
\<	beginning of word
\>	end of word
[ <i>str</i> ]	any char in <i>str</i>
[^ <i>str</i> ]	... not in <i>str</i>
[ <i>x-y</i> ]	... between <i>x</i> and <i>y</i>
*	any number of preceding



---

## Using the `ed` Line Editor

Using the <code>ed</code> Line Editor .....	99
4.1. Getting Started .....	99
Creating Text — the Append Command <code>a</code> .....	100
Error Messages — <code>?</code> .....	101
Writing Text Out as a File — the Write Command <code>w</code> .....	101
Leaving <code>ed</code> — the Quit Command <code>q</code> .....	102
Creating a New File — the Edit Command <code>e</code> .....	102
Exercise: Trying the <code>e</code> Command .....	103
Checking the Filename — the Filename Command <code>f</code> .....	104
Reading Text from a File — the Read Command <code>r</code> .....	104
Printing the Buffer Contents — the Print Command <code>p</code> .....	105
Exercise: Trying the <code>p</code> Command .....	106
Displaying Text — the List Command <code>l</code> .....	106
The Current Line — ‘Dot’ or ‘.’ .....	107
Deleting Lines — the Delete Command <code>d</code> .....	108
Exercise: Experimenting .....	109
Modifying Text — the Substitute Command <code>s</code> .....	109
The Ampersand <code>&amp;</code> .....	111
Exercise: Trying the <code>s</code> and <code>g</code> Commands .....	112
Undoing a Command — the Undo Command <code>u</code> .....	112
4.2. Changing and Inserting Text — the <code>c</code> and <code>i</code> Commands .....	113
Exercise: Trying the <code>c</code> Command .....	113
4.3. Specifying Lines in the Editor .....	114

Context Searching .....	114
Exercise: Trying Context Searching .....	115
Specifying Lines with Address Arithmetic — + and .....	115
Repeated Searches — // and ?? .....	117
Default Line Numbers and the Value of Dot .....	117
Combining Commands — the Semicolon ; .....	119
Interrupting the Editor .....	120
4.4. Editing All Lines — the Global Commands g and v .....	121
Multi-line Global Commands .....	122
4.5. Special Characters .....	123
Matching Anything — the Dot ‘.’ .....	123
Specifying Any Character — the Backslash ‘\’ .....	124
Specifying the End of Line — the Dollar Sign \$ .....	126
Specifying the Beginning of the Line — the Circumflex ^ .....	127
Matching Anything — the Star * .....	127
Character Classes — Brackets [ < ] .....	129
4.6. Cutting and Pasting with the Editor .....	130
Moving Lines Around .....	130
Moving Text Around — the Move Command m .....	130
Substituting Newlines .....	132
Joining Lines — the Join Command j .....	132
Rearranging a Line with <\ ( . . . \) .....	133
Marking a Line — the Mark Command k .....	133
Copying Lines — the Transfer Command t .....	134
4.7. Escaping to the Shell with ! .....	134
4.8. Supporting Tools .....	135
Editing Scripts .....	135
Matching Patterns with grep .....	135
4.9. Summary of Commands and Line Numbers .....	136

---

## Using the ed Line Editor

This chapter<sup>17</sup> describes the editing tools of the ed line editor. It provides the newcomer with elementary instructions and exercises for learning the most necessary and common commands and the more advanced user with information about additional editing facilities. The contents include descriptions of appending, changing, deleting, moving, copying and inserting lines of text; reading and writing files; displaying your files; context searching; the global commands; line addressing; and using special characters. There are also brief discussions on writing scripts and on the pattern-matching tool `grep`, which is related to ed.

We assume that you know how to log in to the system and that you have an understanding of what a file is. You must also know what character to type as the end-of-line on your workstation or terminal. This character is the RETURN key in most cases.

Do the exercises in this chapter as you read along. What you enter at the keyboard is shown in **bold typewriter font like this**.

If you need basic information on the Sun system, refer to the *Beginner's Guide to the Sun Workstation*. See ed in the *Commands Reference Manual for the Sun Workstation* for a nutshell description of the ed commands.

### 4.1. Getting Started

The ed text editor is an interactive program for creating and modifying text, using directions that you provide from your workstation. The text can be a document, a program or perhaps data for a program.

We'll assume that you have logged in to your system, and it is displaying the hostname and prompt character, which we show throughout this manual as:

```
hostname%
```

To use ed, type ed and a carriage return at the 'hostname%' prompt:

```
hostname% ed
hostname%
```

You are now ready to go. ed does not prompt you for information, but waits for you to tell it what to do. First you'll learn how to get some text into a file and later how to change it and make corrections.

---

<sup>17</sup> The material in this chapter is derived from *A Tutorial Introduction to the UNIX Text Editor*, B.W. Kernighan, B.W. Kernighan, Bell Laboratories, Murray Hill, New Jersey.

## Creating Text — the Append Command `a`

Let's assume you are typing the first draft of a memo and starting from scratch. When you first start `ed`, in this case, you are working with a 'blank piece of paper'; there is no text or information present. To supply this text, you either type it in or read it in from a file. To type it in, use the *append* command `a`.

So, to type in lines of text into the buffer, you type an `a` followed by a RETURN, followed by the lines of text you want, like this:

```
hostname% ed
a<CR>
Now is the time
for all good men
to come to the aid of their party.
```

If you make a mistake, use the DEL key to back up over and correct your mistakes. You cannot go back to a previous line after typing RETURN to correct your errors. The only way to stop appending is to tell `ed` that you have finished by typing a line that contains only a period. It takes practice to remember it, but it has to be there. If `ed` seems to be ignoring you, type an extra line with just `'.'` on it. You may then find you've added some garbage lines to your text; you will have to take them out later.

After the append command, your file contains the lines:

```
Now is the time
for all good men
to come to the aid of their party.
```

The `a` and `'.'` aren't there, because they are not text.

To add more text to what you already have, type another `a`, and continue typing.

If you have not used a text editor before, read the following to learn a bit of terminology. If you have used an editor, skip to the "Error Messages — ?" section.

In `ed` jargon, the text being worked on is said to be in a work space or 'kept in a buffer'. The buffer is like a piece of paper on which you write things, change some of them, and finally file the whole thing away for another day.

You have learned how to tell `ed` what to do to the text by typing instructions called *commands*. Most commands consist of a single letter that you type in lower case letters. An example is the append command `a`. Type each command on a separate line. You sometimes precede the command by information about what line or lines of text are to be affected; we discuss this shortly.

As you have seen, `ed` does not respond to most commands; that is, there isn't any prompting or message display like 'ready'. If this bothers you as a beginner, be patient. You'll get used to it.

## Error Messages — ?

When you make an error in the commands you type, ed asks you:

```
?
```

This is about as cryptic as it can be, but with practice, you can usually figure out how you goofed.

Writing Text Out as a File —  
the Write Command w

When you want to save your text for later use, write out the contents of the buffer into a file with the *write* command `w`, followed by the filename you want to write in. The `w` command copies the buffer's contents into the specified file, destroying any previous information on the file. To save the text in a file named *junk*, for example, type:

```
w junk
68
```

Leave a space between the `w` and the filename. ed responds by displaying the number of characters it wrote out, in this case 68. Remember that blanks and the return character at the end of each line are included in the character count. The buffer's contents are not disturbed, so you can go on adding lines to it. This is an important point. ed works on a copy of a file at all times, not on the file itself. There is no change in the contents of a file until you type a `w`. Writing out the text into a file from time to time is a good idea to save most of your text should you make some horrible mistake. If you do something disastrous, you only lose the text in the buffer, not the text that was written into the file.

When you want to copy a portion of a file to another name so you can format it separately, use the `w` command. Suppose that in the file being edited you have:

```
.TS
...lots of stuff
.TE
```

This is the way a table is set up for the `tbl` program. To isolate the table in a separate file called, for example, *table*, first find the start of the table (the `.TS` line), then write out the interesting part:

```
/^\.TS/
.TS (ed prints the line it found)
.,/^\.TE/w table
```

and the job is done. If you are confident, you can do it all at once with:

```
/^\.TS/;/^\.TE/w table
```

The point is that `w` can write out a group of lines, instead of the whole file. In fact, you can write out a single line if you like; give one line number instead of two (we explain line numbers later — see the section "Specifying Lines in the Editor" for details). For example, if you have just typed a very long, complicated line and you know that you are going to need it or something like it later, then save it — don't re-type it. In the editor, say:

```
a
...lots of stuff...
...very long, complicated line...
.
.w temp
number of characters
a
...more stuff...
.
.r temp
number of characters
a
...more stuff...
.
```

This last example is worth studying to be sure you appreciate what's going on. The `.w temp` writes the very long, complicated line (the current line) you typed to the file called `temp`. The `.r temp` reads that line from `temp` into the file you are editing after the current line 'dot' so you don't have to re-type it.

#### Leaving `ed` — the Quit Command `q`

To terminate an `ed` session, save the text you're working on by writing it into a file using the `w` command, and then type the *quit* command `q`.

```
w
number of characters
q
hostname%
```

The system responds with the hostname prompt. At this point your buffer vanishes, with all its text, which is why you want to write it out before quitting. Actually, `ed` displays '?' if you try to quit without writing. At that point, write the file if you want; if not, type another `q` to get you out of `ed` regardless of whether you changed the file or not.

#### Creating a New File — the Edit Command `e`

The `edit` command `e` says 'I want to edit a new file called *newfile*, without leaving the editor.' To do this, you type:

```
e newfile
```

The `e` command discards whatever you're currently working on and starts over on *newfile*. It's exactly the same as if you had quit with the `q` command, then re-entered `ed` with a new filename, except that if you have a pattern remembered, a command like `//` will still work. (See the section "Repeated Searches — `//` and `??`" later in this chapter.)

If you enter `ed` with the command:

```
hostname% ed file
```

`ed` remembers the name of the file, and any subsequent `e`, `r` or `w` commands that don't contain a filename refer to this remembered file. Thus:

```

hostname% ed file1
... (editing) ...
w      (writes back in file1)
e file2 (edit new file, without leaving editor)
... (editing in file2) ...
w      (writes back in file2)

```

and so on does a series of edits on various files without ever leaving `ed` and without typing the name of any file more than once.

A common way to get text into the buffer is to read it from a file in the file system. This is what you do to edit text that you saved with `w` in a previous session. The `edit` command `e` also fetches the entire contents of a file into the buffer. So if you had saved the three lines ‘Now is the time’, etc., with `w` in an earlier session, the `ed` command `e` fetches the entire contents of the file `junk` into the buffer, and responds with the number of characters in `junk`:

```

hostname% e junk
68

```

*If anything was already in the buffer, it is deleted first.*

If you use `e` to read a file into the buffer, you do not need to use a filename after a subsequent `w` command; `ed` remembers the last filename used in an `e` command, and `w` will write on this file. Thus a good way to operate is:

```

hostname% ed
e file
number of characters
[editing session]
w
number of characters
q
hostname%

```

This way, you can simply say `w` from time to time, and be secure that you are writing into the proper file each time.

### Exercise: Trying the `e` Command

Experiment with the `e` command — try reading and displaying various files. You may get an error

```
?name
```

where *name* is the name of a file; this means that the file doesn’t exist, typically because you spelled the filename wrong, or perhaps because you are not allowed to read or write it. Try alternately reading and appending to see that they work similarly. Verify that:

```

hostname% ed filename
number of characters in file

```

is equivalent to:

```
hostname% ed
e filename
number of characters in file
```

### Checking the Filename — the Filename Command `f`

You can find out the remembered filename at any time with the `f` command; just type `f` without a filename. In this example, if you type `f`, `ed` replies:

```
hostname% ed junk
68
f
junk
```

You can also change the name of the remembered filename with `f`; this following sequence guarantees that a careless `w` command will write on *junk* instead of *precious*. Try:

```
hostname% ed precious
f junk
... (editing) ...
```

### Reading Text from a File — the Read Command `r`

Sometimes you want to read a file into the buffer without destroying anything that is already there. To do this, use the *read* command `r`. The command:

```
r junk
68
```

reads the file *junk* into the buffer, adding it to the end of whatever is already in the buffer. `ed` responds with the number of characters in the buffer. So if you do a read after an edit:

```
hostname% ed junk
68
r junk
68
w
136
q
hostname%
```

the buffer contains *two* copies of the text or six lines (136 characters) in this case. Like `w` and `e`, `r` displays the number of characters read in after the reading operation is complete. Now check the file contents with `cat`:

```
hostname% cat junk
Now is the time
for all good men
to come to the aid of their party.
Now is the time
for all good men
to come to the aid of their party.
hostname%
```

Generally speaking, you won't use `r` as much as `e`.

Suppose you have a file called *memo*, and you want the file called *table* to be inserted just after the reference to Table 1. That is, in *memo* somewhere is a line that says

Table 1 shows that ...

The data contained in *table* has to go there so `nroff` or `troff` will format it properly. Now what?

This one is easy. Edit *memo*, find 'Table 1', and add the file *table* right there:

```
hostname% ed memo
/Table 1/
Table 1 shows that ... (response from ed)
.r table
```

The critical line is the last one. As we said earlier, the `r` command reads a file; here you asked for it to be read in right after line dot. An `r` command without any address adds lines at the end, which is the same as `$r`.

## Printing the Buffer Contents — the Print Command `p`

To print or 'display' the contents of the buffer or parts of it on the screen, use the `print` command `p`. To do this, specify the lines where you want the display to begin and where you want it to end, separated by a comma, and followed by `p`. Thus to show the first two lines of the buffer, for example, say:

```
1, 2p (starting line=1, ending line=2 p)
Now is the time
for all good men
```

Suppose you want to print *all* the lines in the buffer. You could use `1, 3p` if you knew there were exactly three lines in the buffer. But in general, you don't know how many lines there are, so what do you use for the ending line number? `ed` provides a shorthand symbol for 'line number of last line in buffer' — the dollar sign `$`. Use it to display *all* the lines in the buffer, line 1 to last line:

```
1, $p
Now is the time
for all good men
to come to the aid of their party.
Now is the time
for all good men
to come to the aid of their party.
```

If you want to stop the display of more than one screenful before it is finished, type the INTERRUPT character CTRL-C (or the DEL key).

```
CTRL-C
?
```

`ed` waits for the next command.

To display the *last* line of the buffer, you can use:

**\$, \$p**

to come to the aid of their party.

or abbreviate it to:

**\$p**

to come to the aid of their party.

You can show any single line by typing the line number followed by a `p`. So, to display the first line of the buffer, type:

**1p**

Now is the time

In fact, `ed` lets you abbreviate even further: you can display any single line by typing *just* the line number — there is no need to type the letter `p`. So if you say:

**2**

for all good men

`ed` displays the second line of the buffer.

You can also use `$` in combinations to display the last two lines of the buffer, for example:

**\$(-1, \$p**

for all good men

to come to the aid of their party.

This helps when you want to see how far you got in typing.

### Exercise: Trying the `p` Command

As before, create some text using the `a` command and experiment with the `p` command. You will find, for example, that you can't show line 0 or a line beyond the end of the buffer, and that attempts to show a buffer in reverse order don't work. For example, you get an error message if you type:

**3, 1p**

?

### Displaying Text — the List Command `l`

`ed` provides two commands for displaying the contents of the lines you're editing. You are familiar with the `p` command that displays lines of text. Less familiar is the *list* command `l` (the letter 'ell'), which gives slightly more information than `p`. In particular, `l` makes visible characters that are normally invisible, such as tabs and backspaces. If you list a line that contains some of these, `l` will show each tab as `>` and each backspace as `<`. A sample display of a random file with tab characters and backspaces is:

**l**

Now is the >> time for << all good men

This makes it much easier to correct the sort of typing mistake that inserts extra spaces adjacent to tabs, or inserts a backspace followed by a space.

The `l` command also ‘folds’ long lines for printing. Any line that exceeds 72 characters is displayed on multiple lines. Each printed line except the last is terminated by a backslash ‘\’, so you can tell it was folded. This is useful for displaying long lines on small terminal screens. A sample output of a folded line is:

```
1
This is an example of using the l command to display a very long line that \
has more than 72 characters ...
```

Occasionally the `l` command displays in a line a string of numbers preceded by a backslash, such as ‘\07’ or ‘\16’. These combinations make visible the characters that normally don’t show, like form feed or vertical tab or bell. Each such combination is a single character. When you see such characters, be wary — they may have surprising meanings when displayed on some terminals. Often their presence means that your finger slipped while you were typing; you almost never want them.

### The Current Line — ‘Dot’ or ‘.’

Suppose your buffer still contains the six lines as above, and that you have just typed:

```
1, 3p
Now is the time
for all good men
to come to the aid of their party.
```

ed has displayed the three lines for you. Try typing just a `p` to display:

```
p (no line numbers)
to come to the aid of their party.
```

The line displayed is the third line of the buffer. In fact it is the last or most recent line that you have done anything with. (You just displayed it!) You can repeat `p` without line numbers, and it will continue to display line 3.

The reason is that ed maintains a record of the last line that you did anything to (in this case, line 3, which you just displayed) so that you can use it instead of an explicit line number. You refer to this most recent line by the shorthand symbol:

```
. (pronounced ‘dot’)
to come to the aid of their party.
```

Dot is a line number in the same way that ‘\$’ is; it means exactly ‘the current line’, or loosely, ‘the line you most recently did something to’. You can use it in several ways — one possibility is to display all the lines from and including the current line to the end of the buffer.

```
., $p
Now is the time
for all good men
to come to the aid of their party.
to come to the aid of their party.
```

In our example these are lines 3 through 6.

Some commands change the value of dot, while others do not. The `p` command sets dot to the number of the last line displayed; that is, after this command sets both `'.'` and `'$'` refer to the last line of the file, line 6.

Dot is most useful in combinations like:

`+.1` (or equivalently, `+.1p`)

This means 'show the next line' and is a handy way to step slowly through a buffer. You can also say:

`-.1` (or `-.1p`)

This means 'show the line *before* the current line'. Use this to go backward if you wish. Another useful one is something like:

`-.3, -.1p`

This command displays the previous three lines.

Don't forget that all of these change the value of dot. You can find out what dot is at any time by typing:

```
.=  
3
```

Let's summarize some things about `p` and dot. Essentially you can precede `p` by 0, 1, or 2 line numbers. If you do not give a line number, `p` shows the 'current line', the line that dot refers to. If there is one line number given with or without the letter `p`, it shows that line and dot is set there; and if there are two line numbers, it shows all the lines in that range, and sets dot to the last line displayed. If you specify two line numbers, the first can't be bigger than the second.

Typing a single RETURN displays the next line — it's equivalent to `+.1p`. Try it. Try typing a `-`; you will find that it's equivalent to `-.1p`.

## Deleting Lines — the Delete Command `d`

Suppose you want to get rid of the three extra lines in the buffer. To do this, use the delete command `d`. The `d` command is similar to `p`, except that `d` deletes lines instead of displaying them. You specify the lines to be deleted for `d` exactly as you do for `p`:

*starting line, ending line* `d`

Thus the command:

`4, $d`

deletes lines 4 through the end. There are now three lines left, as you can check by using:

`1, $p`

```
Now is the time  
for all good men  
to come to the aid of their party.
```

And notice that `'$'` now is line 3. Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot

is set to '\$'.

### Exercise: Experimenting

Experiment with `a`, `e`, `r`, `w`, `p` and `d` until you are sure you know what they do, and until you understand how to use dot, '\$' and the line numbers.

If you are adventurous, try using line numbers with `a`, `r` and `w` as well. You will find that `a` appends lines *after* the line number that you specify rather than after dot; that `r` reads a file in *after* the line number you specify and not necessarily at the end of the buffer; and that `w` writes out exactly the lines you specify, not necessarily the whole buffer. These variations are useful, for instance, for inserting a file at the beginning of a buffer:

```
0x filename
   number of characters
```

ed indicates the number of characters read in. You can enter lines at the beginning of the buffer by saying:

```
0a
. . . text . . .
.
```

Or you can write out the lines you specify with `w`. Notice that `.w` is very different from:

```
.
w
   number of characters
```

### Modifying Text — the Substitute Command `s`

One of the most important commands is the *substitute* command `s`. Use `s` to change individual words or letters within a line or group of lines. For example, you can correct spelling mistakes and typing errors.

Suppose that by a typing error, line 1 says:

```
Now is th time
```

— the 'e' has been left off 'the'. You can use `s` to fix this up as follows:

```
1s/th/the/
```

This says: 'in line 1, substitute for the characters 'th' the characters 'the'. ed does not display the result automatically, so verify that it works with:

```
P
Now is the time
```

You get what you wanted. Notice that dot has been set to the line where the substitution took place, since `p` printed that line. The `s` command always sets dot in this way.

The general way to use the substitute command is:

```
starting-line, ending-line s/change this/to this/
```

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in *all* the lines between *starting-line* and

*ending-line* . Only the first occurrence on each line is changed, however. If you want to change *every* occurrence, read on below. The rules for line numbers are the same as those for `p`, except that dot is set to the last line changed. But there is a trap for the unwary: if no substitution took place, dot is *not* changed. This causes an error '?' as a warning.

Thus you can say:

```
1, $s/speling/spelling/
```

and correct the first spelling mistake on each line in the text. (This is useful for people who are consistent misspellers!)

You can precede any `s` command by one or two 'line numbers' to specify that the substitution is to take place on a group of lines. Thus, to change the *first* occurrence of 'mispell' to 'misspell' on every line of the file, type:

```
1, $s/mispell/misspell/
```

But to change *every* occurrence in every line, type:

```
1, $s/mispell/misspell/g
```

This is more likely what you wanted in this particular case.

*Note:* Be careful that this is exactly what you want to do. Unless you specify the substitution specifically, globally changing the string 'the', will also change every instance of those characters, including 'other', etc.

If you do not give any line numbers, `s` assumes you mean 'make the substitution on line dot,' so it changes things only on the current line. You will see that a very common sequence is to correct a mistake on the current line, and then display the line to make sure everything is all right:

```
s/something/something else/p  
line with something else
```

If it didn't, you can try again.

Notice that there is a `p` on the same line as the `s` command. With few exceptions, `p` can follow any command. *No other multi-command lines are legal.*

You can also say:

```
s/ ... //
```

which means 'change the first string of characters to *nothing*;' that is, remove the first string of characters. Use this sequence for deleting extra words in a line or removing extra letters from words. For instance, if you had:

```
Nowxx is the time
```

To correct this, say:

```
s/xx//p  
Now is the time
```

Notice that `//` (two adjacent slashes) means 'no characters,' not a blank. There is a difference! (See the section "Repeated Searches" for another meaning of `//`.)

If you want to replace the *first* 'this' on a line with 'that', for example, use:

```
s/this/that/
```

If there is more than one 'this' on the line, a second form with the trailing *global* command `g` changes *all* of them:

```
s/this/that/g
```

The general format is:

```
s/... /... /gp
```

Try other characters instead of slashes to delimit the two sets of characters in the `s` command — anything should work except blanks or tabs. If you get funny results using any of the characters:

```
^ . $ [ * \ &
```

read the section on "Special Characters".

You can follow either form of the `s` command by `p` or `l` to display or list the contents of the line.

```
s/this/that/p
s/this/that/l
s/this/that/gp
s/this/that/gl
```

are all acceptable and mean slightly different things. Make sure you know what the differences are.

You should also notice that if you add a `p` or `l` to the end of any of these substitute commands, only the last line that was changed will be displayed, not all the lines. We will talk later about how to show all the lines that were modified.

## The Ampersand &

The `&` is a shorthand character — it is used only on the right-hand part of a substitute command where it means 'whatever was matched on the left-hand side'. Use it to save typing. Suppose the current line contained:

```
Now is the time
```

and you wanted to put parentheses around it. You could just retype the line, but this is tedious. Or you could say:

```
s/^/(/
s/$)/)
```

using your knowledge of `^` and `$`. But the easiest way uses the `&`:

```
s/.*/(&)/
```

This says 'match the whole line, and replace it by itself surrounded by parentheses'.

You can use the `&` several times in a line:

```
s/.*/&? &!!!/
Now is the time? Now is the time!!
```

or

```
s/the/& best and & worst/  
Now is the best and the worst time
```

You don't have to match the whole line, of course, if the buffer contains:

```
the end of the world
```

you can type:

```
/world/s//& is at hand/  
the end of the world is at hand
```

Observe this expression carefully, for it illustrates how to take advantage of `s` to save typing. The string `/world/` found the desired line; the shorthand `//` found the same word in the line; and the `&` saves you from typing it again.

Notice that `&` is not special on the left side of a substitute, only on the *right* side.

The `&` is a special character only within the replacement text of a substitute command, and has no special meaning elsewhere. You can turn off the special meaning of `&` by preceding it with a backslash (`\`):

```
s/ampersand/\&/
```

converts the word 'ampersand' into the literal symbol '&' in the current line. Of course this isn't much of a saving if the thing matched is just 'the', but if it is something truly long or awful, or if it is something like `.*` which matches a lot of text, you can save some tedious typing. There is also much less chance of making a typing error in the replacement text. For example, to put parentheses around a line, regardless of its length, use:

```
s/.*/(&)/
```

### Exercise: Trying the `s` and `g` Commands

Experiment with `s` and `g`. See what happens if you substitute for some word on a line with several occurrences of that word. For example, do this:

```
a  
the other side of the coin  
.br/>s/the/on the/p  
on the other side of the coin
```

### Undoing a Command — the Undo Command `u`

Occasionally you will make a substitution in a line, only to realize too late that it was a mistake. Use the *undo* command `u` to undo the last substitution. This restores the last line that was substituted to its previous state. For example, study the following example:

```
s/party/country/
P
to come to the aid of their country.
u
P
to come to the aid of their party.
```

#### 4.2. Changing and Inserting Text — the `c` and `i` Commands

This section discusses the *change* command `c` and the *insert* command `i`. The change command changes or replaces a group of one or more lines. The insert command inserts a group of one or more lines.

The `c` command replaces a number of lines with different lines you type in at the workstation. For example, to change lines `+.1` through `$` to something else, type:

```
.+1,$c
. . . type the lines of text you want here . . .
.
```

The lines you type between the `c` command and the `.` take the place of the original lines between start line and end line. This is most useful in replacing a line or several lines that have errors in them.

If you only specify one line in the `c` command, just that line is replaced. You can type in as many replacement lines as you like. Notice the use of `.` to end the input — this works just like the `.` in the append command and must appear by itself on a new line. If no line number is given, line dot is replaced. The value of dot is set to the last line you typed in.

‘Insert’ is similar to append, for instance:

```
/string/i
. . . type the lines to be inserted here . . .
.
```

inserts the given text *before* the next line that contains ‘string’, that is, the text between `i` and `.` is inserted *before* the specified line. If no line number is specified dot is used. Dot is set to the last line inserted.

#### Exercise: Trying the `c` Command

Change is rather like a combination of delete followed by insert. Experiment to verify that:

```
start,end d
i
...text...
.
```

is almost the same as:

```
start,end c
...text...
.
```

These are not *precisely* the same if line `$` gets deleted. Check this out. What is dot?

Experiment with `a` and `i`, to see that they are similar, but not the same. You will observe that to append *after* the given line, you type:

```
line-number a
...text...
```

while to insert *before* it, you type:

```
line-number i
...text . . .
```

Observe that if you do not give a line number, `i` inserts before line dot, while `a` appends after line dot.

### 4.3. Specifying Lines in the Editor

To specify which lines are to be affected by the editing commands, you use *line addressing*. There are several methods, and they are described below.

#### Context Searching

One way is *context searching*. Context searching is simply a method of specifying the desired line, regardless of what its number is, by specifying some context on it.

Suppose you have the original three-line text in the buffer:

```
Now is the time
for all good men
to come to the aid of their party.
```

If you want to find the line that contains 'their' so you can change it to 'the'. With only three lines in the buffer, it's pretty easy to keep track of what line the word 'their' is on. But if the buffer contains several hundred lines, and you'd been making changes, deleting and rearranging lines, and so on, you would no longer really know what this line number would be.

For example, to locate the next occurrence of the characters between slashes ('their'), type:

```
/their/
to come to the aid of their party.
```

To search for a line that contains a particular string of characters, the general format is:

```
/string of characters we want to find/
```

This is sufficient to find the desired line. It also sets dot to that line and displays the line for verification. 'Next occurrence' means that `ed` starts looking for the string at line `+.1`, searches to the end of the buffer, then continues at line 1 and searches to line dot. That is, the search 'wraps around' from '\$' to 1. It scans all the lines in the buffer until it either finds the desired line or gets back to dot again. If the given string of characters can't be found in any line, `ed` displays the error message:

```
?
```

Otherwise it shows the line it found.

Less familiar is the use of:

`?thing?`

This command scans *backward* for the previous occurrence of 'thing'. This is especially handy when you realize that the thing you want to operate on is back up the page from where you are currently editing.

The slash and question mark are the only characters you can use to delimit a context search, though you can use essentially any character in a substitute command. You can do both the search for the desired line *and* a substitution all at once, like this:

```
/their/s/their/the/p
to come to the aid of the party.
```

There were three parts to that last command: a context search for the desired line, the substitution, and displaying the line.

The expression `/their/` is a context search expression. In their simplest form, all context search expressions are like this — a string of characters surrounded by slashes. Context searches are interchangeable with line numbers, so you can use them by themselves to find and show a desired line, or as line numbers for some other command, like `s`. We use them both ways in the examples above.

### Exercise: Trying Context Searching

Experiment with context searching. Try a body of text with several occurrences of the same string of characters, and scan through it using the same context search.

Try using context searches as line numbers for the substitute, print and delete commands. You can also use context searching with `r`, `w`, and `a`.

If you get funny results with any of the characters:

```
^ . $ [ * \ &
```

read the section on "Special Characters".

### Specifying Lines with Address Arithmetic — + and -

Another area where you can save typing in specifying lines is to use minus (-) and plus (+) as line numbers by themselves. To move back up one line in the file, type:

```
-
```

In fact, you can string several minus signs together to move back up that many lines:

```
---
```

moves up three lines, as does `-3`. Thus:

```
-3, 3p
```

is also identical to the examples above.

Since `-` is shorter than `.-1`, use it to change 'bad' to 'good' on the previous line and on the current line.

```
- , .s/bad/good/
```

You can use `+` and `-` in combination with searches using `/.../` and `?...?`, and with `$`. To find the line containing 'thing', and position you two lines before it, type:

```
/thing/--
```

The next step is to combine the line numbers like `'.'` and `'$'`, context searches like `'/.../'` and `'?...?'` with `'+'` and `'-'`. Thus:

```
$$-1
```

displays the next-to-last line of the current file, that is, one line before line `'$'`. For example, to recall how far you got in a previous editing session, type:

```
$$-5, $p
```

which shows the last six lines. (Be sure you understand why it shows six, not five.) If there are less than six, of course, you'll get an error message. Suppose the buffer contains the three familiar lines:

```
Now is the time  
for all good men  
to come to the aid of their party.
```

Then the ed line numbers:

```
/Now/+1  
/good/  
/party/-1
```

are all context search expressions, and they all refer to the same line, line 2. To make a change in line 2, you could say:

```
/Now/+1s/good/bad/
```

or:

```
/good/s/good/bad/
```

or:

```
/party/-1s/good/bad/
```

Convenience dictates the choice. You could display all three lines by, for instance:

```
/Now/, /party/p
```

or:

```
/Now/, /Now/+2p
```

or by any number of similar combinations. The first one of these might be better if you don't know how many lines are involved. Of course, if there were only three lines in the buffer, you'd use:

```
1, $p
```

but not if there were several hundred.

The basic rule is: a context search expression is *the same as* a line number, so you can use it wherever a line number is needed.

As another example:

```
.-3, .+3p
```

displays from three lines before where you are now at line dot to three lines after, thus giving you a bit of context. By the way, you can omit the '+':

```
.-3, .3p
```

is identical in meaning.

### Repeated Searches — // and ??

Suppose you ask for the search:

```
/horrible thing/
```

and when the line is displayed, you discover that it isn't the horrible thing that you wanted, so you have to repeat the search again. You don't have to re-type the search; use the construction:

```
//
```

as a shorthand for 'the previous thing that was searched for', whatever it was. You can repeat this as many times as necessary. You can also search backward through the file by typing:

```
??
```

?? searches for the same thing, but in the reverse direction.

Not only can you repeat the search, but you can use '/' as the left side of a substitute command, to mean 'the most recent pattern.'

```
/horrible thing/  
... ed prints line with 'horrible thing' ...  
s//good/p
```

To go backward and change a line, say:

```
??s//good/
```

You can still use the & on the right hand side of a substitute to stand for whatever got matched:

```
//s//& &/p
```

This finds the next occurrence of whatever you searched for last, replaces it by two copies of itself, then displays the line just to verify that it worked.

### Default Line Numbers and the Value of Dot

One of the most effective ways to speed up your editing is always to know what lines will be affected by a command if you don't specify the lines it is to act on, and on what line you will be positioned, that is, the value of dot, when a command finishes. If you can edit without specifying unnecessary line numbers, you can save a lot of typing.

As the most obvious example, if you give a search command like:

```
/thing/
```

you are left pointing at the next line that contains 'thing'. No address is required with commands like `s` to make a substitution on that line. Addresses are also not required with `p` to show it, `l` to list it, `d` to delete it, `a` to append text after it, `c` to change it, or `i` to insert text before it.

What would happen if there were no 'thing'? Then you are left right where you were — dot is unchanged. This is also true if you are sitting on the only 'thing' when you issue the command. The same rules hold for searches that use `? . . . ?`; the only difference is the direction in which you search.

The delete command `d` leaves dot pointing at the line that followed the last deleted line. When line '\$' gets deleted, however, dot points at the *new* line '\$'.

The line-changing commands `a`, `c` and `i` by default all affect the current line. If you do not give a line number with them, the `a` appends text after the current line, `c` changes the current line, and `i` inserts text before the current line.

The `a`, `c`, and `i` commands behave identically in one respect — when you stop appending, changing or inserting, dot points at the last line entered. This is exactly what you want for typing and editing on the fly. For example, you can say:

```
a
... text ...
... botch ...           (minor error)
.                       (to get out of append mode)
s/botch/correct/      (fix botched line)
a
... more text ...
```

without specifying any line number for the substitute command or for the second append command. Or you can say:

```
a
... text ...
... horrible botch ... (major error)
.                       (to get out of append mode)
c                       (replace entire line)
... fixed up line ...
```

You should experiment to determine what happens if you do not add *any* lines with `a`, `c` or `i`.

The `r` command reads a file into the text being edited, either at the end if you do not give an address, or after the specified line if you do. In either case, dot points at the last line read in. Remember that you can even say `0r` to read a file in at the beginning of the text. You can also say `0a` or `1i` to start adding text at the beginning.

The `w` command writes out the entire file. If you precede the command by one line number, that line is written, while if you precede it by two line numbers, that range of lines is written. The `w` command does *not* change dot; the current line

remains the same, regardless of what lines are written. This is true even if you say something that involves a context search, such as:

```
/^\.AB/,/^\.AE/w abstract
```

Since `w` is so easy to use, you should save what you are editing regularly as you go along just in case something goes wrong, or in case you do something foolish, like clobbering what you're editing.

With the `s` command, the rule is simple; you are left positioned on the last line that got changed. If there were no changes, dot doesn't move.

To illustrate, suppose that there are three lines in the buffer, and the cursor is sitting on the middle one:

```
x1
x2
x3
```

The command line

```
-,+s/x/y/p
```

displays the third line, the last one changed. But if the three lines had been:

```
x1
y2
y3
```

and the same command had been issued while dot pointed at the second line, then the result would be to change and show only the first line, and that is where dot would be set.

### Combining Commands — the Semicolon ;

Searches with `/.../` and `?...?` start at the current line and move forward or backward respectively until they either find the pattern or get back to the current line. Sometimes this is not what is wanted. Suppose, for example, that the buffer contains lines like this:

```
.
.
.
ab
.
.
bc
.
.
```

Starting at line 1, one would expect that the command:

```
/a/,/b/p
```

would display all the lines from the 'ab' to the 'bc' inclusive. Actually this is not what happens. *Both* searches (for 'a' and for 'b') start from the same point, and thus they both find the line that contains 'ab'. The result is to display a single line. Worse, if there had been a line with a 'b' in it before the 'ab' line, then the

print command would be in error, since the second line number would be less than the first, and you cannot display lines in reverse order.

This happens because the comma separator for line numbers doesn't set dot as each address is processed; each search starts from the same place. In `ed`, you can use the semicolon `;` just like comma, with the single difference that use of a semicolon forces dot to be set at that point as the line numbers are being evaluated. In effect, the semicolon 'moves' dot. Thus in the example above, the command:

```
/a;/b/p
```

displays the range of lines from 'ab' to 'bc', because after the 'a' is found, dot is set to that line, and then 'b' is searched for, starting beyond that line.

Use the semicolon when you want to find the *second* occurrence of something. For example, to find the second occurrence of 'thing', you can say:

```
/thing/
line with 'thing'
//
second line with 'thing'
```

But this displays the first occurrence as well as the second, and is a nuisance when you know very well that it is only the second one you're interested in. The solution is to find the first occurrence of 'thing', set dot to that line, then find the second and display only that:

```
/thing//;
```

Closely related is searching for the second previous occurrence of something, as in:

```
?something?;??
```

We leave you to try showing the third or fourth or ... in either direction.

Finally, bear in mind that if you want to find the first occurrence of something in a file, starting at an arbitrary place within the file, it is not sufficient to say:

```
1;/thing/
```

This search fails if 'thing' occurs on line 1. But it is possible to say:

```
0;/thing/
```

This is one of the few places where 0 is a legal line number, for this starts the search at line 1.

## Interrupting the Editor

As a final note on what dot gets set to, be aware that if you type an INTERRUPT (CTRL-C is the default, but your terminal may be set up with the DELETE, RUBOUT or BREAK keys) while `ed` is doing a command, things are put back together again and your state is restored as much as possible to what it was before the command began. Naturally, some changes are irrevocable — if you are reading or writing a file or making substitutions or deleting lines, these will be stopped in the middle of execution in some clean but unpredictable state; hence it is not usually wise to stop them. Dot may or may not be changed.

Displaying is more clear cut. Dot is not changed until the display is done. Thus if you display lines until you see an interesting one, then type CTRL-C, you are *not* sitting on that line or even near it. Dot is left where it was when the `p` command was started.

#### 4.4. Editing All Lines — the Global Commands `g` and `v`

Use the *global* command `g` to execute one or more `ed` commands on all those lines in the buffer that match some specified string. For example, to display all lines that contain 'peling', type:

```
g/peling/p
```

As another example:

```
g/^\./p
```

displays all the formatting commands in a file. The pattern that goes between the slashes can be anything that could be used in a line search or in a substitute command; the same rules and limitations apply.

For a more useful command, which makes the substitution everywhere on the line, then displays each corrected line, type:

```
g/peling/s//pelling/gp
```

Compare this to the following command line, which only displays the last line substituted:

```
1,$s/peling/pelling/gp
```

Another subtle difference is that the `g` command does not give a '?' if 'peling' is not found whereas the `s` command will.

The substitute command is probably the most useful command that can follow a global because you can use this to make a change and display each affected line for verification. For example, you can change the word 'SUN' to 'Sun' everywhere in a file, and verify that it really worked, with:

```
g/SUN/s//Sun/gp
```

Notice that you use `//` in the substitute command to mean 'the previous pattern', in this case, 'SUN'. The `p` command is done on every line that matches the pattern, not just those on which a substitution took place.

The `v` command is identical to `g`, except that it operates on those lines that do *not* contain an occurrence of the pattern; that is, `v` 'inverts' the process, so:

```
v/^\./p
```

The command that follows `g` or `v` can be anything:

```
g/^\./d
```

deletes all lines that begin with '.', and:

```
g/^$/d
```

deletes all empty lines.

The global command operates by making two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each marked line in turn is examined, dot is set to that line, and the command executed. This means that it is possible for the command that follows a `g` or `v` to use addresses, set dot, and so on, quite freely.

```
g/^\.PP/+
```

displays the line that follows each `.PP` command (the signal for a new paragraph in some formatting packages). Remember that `+` means 'one line past dot'. And:

```
g/topic/?^\.SH?1
```

searches for each line that contains 'topic', scans backward until it finds a line that begins `.SH` (a section heading) and shows the line that follows that, thus showing the section headings under which 'topic' is mentioned. Finally:

```
g/^\.EQ+/,/^\.EN/-p
```

displays all the lines that lie between lines beginning with `.EQ` and `.EN` formatting commands.

You can also precede the `g` and `v` commands by line numbers, in which case the lines searched are only those in the range specified.

## Multi-line Global Commands

You can use more than one command under the control of a global command, although the syntax for expressing the operation is not especially natural or pleasant. As an example, suppose the task is to change 'x' to 'y' and 'a' to 'b' on all lines that contain 'thing'. Then:

```
g/thing/s/x/y\  
s/a/b/
```

is sufficient. The `\` signals `g` that the set of commands continues on the next line; it terminates on the first line that does not end with `\`. You can't use a substitute command to insert a newline within a `g` command.

Watch out for the command:

```
g/x/s//y\  
s/a/b/
```

which does *not* work as you expect. The remembered pattern is the last pattern that was actually executed, so sometimes it will be 'x' (as expected), and sometimes it will be 'a' (not expected). You must spell it out, like this:

```
g/x/s/x/y\  
s/a/b/
```

It is also possible to execute `a`, `c` and `i` commands under a global command; as with other multi-line constructions, all that is needed is to add an `\` at the end of each line except the last. Thus to add a `.nf` and `.sp` command before each `.EQ` line, type:

```
g/^\.EQ/i\  
.nf\  
.sp
```

You do not need a final line containing a '.' to terminate the `i` command, unless you are using further commands under the global command. On the other hand, it does no harm to put it in either.

#### 4.5. Special Characters

Certain characters have unexpected meanings when they occur in the left side of a substitute command, or in a search for a particular line. You may have noticed that things just don't work right when you use some characters like '.', \*, \$, and others in context searches and with the substitute command. These special characters are called *metacharacters*. Basically, `ed` treats these characters as special, with special meanings. For instance, in a context search or the first string of the substitute command only, '.' means 'any character,' not a period, so:

```
/x.y/
```

means 'a line with an 'x', *any character*, and a 'y', *not* just 'a line with an 'x', a period, and a 'y'.' A complete list of the special characters is:

```
^ . $ [ * \  
_ { } |
```

#### Matching Anything — the Dot '.'

Use the 'dot' metacharacter '.' to match any single character. For example, to find any line where 'x' and 'y' occur separated by a single character, type:

```
/x.y/
```

You may get any of:

```
x+y  
x-y  
x y  
x.
```

and so on.

Since '.' matches a single character, it gives you a way to deal with funny characters that `l` displays. Suppose you have a line that, when displayed with the `l` command, appears as:

```
.... th07is ....
```

and you want to get rid of the 07 (which represents the bell character, by the way).

The most obvious solution is to try:

```
s/07//
```

but this will fail. (Try it.) The brute force solution, which most people would now take, is to re-type the entire line. This is guaranteed, and is actually quite a reasonable tactic if the line in question isn't too big, but for a very long line, re-typing is a bore. This is where the metacharacter '.' comes in handy. Since '07' really represents a single character, if we say:

```
s/th.is/this/
```

the job is done. The '.' matches the mysterious character between the 'h' and the 'i', *whatever it is*.

Bear in mind that since '.' matches any single character, the command:

```
s/./,/
```

converts the first character on a line into a comma (,), which very often is not what you intended.

As is true of many characters in `ed`, the '.' has several meanings, depending on its context. This line shows all three:

```
.s/././
```

The first '.' is a line number, the number of the line we are editing, which is called 'line dot'. The second '.' is a metacharacter that matches any single character on that line. The third '.' is the only one that really is an honest literal period. On the *right* side of a substitution, '.' is not special. If you apply this command to the line:

```
Now is the time.
```

the result will be:

```
.s/././
.now is the time.
```

which is probably not what you intended.

### Specifying Any Character — the Backslash '\'

The backslash character '\' is special to `ed` as noted in the description of the ampersand. For safety's sake, avoid the backslash where possible. If you have to use one of the special characters in a substitute command, you can turn off its magic meaning temporarily by preceding it with the backslash. Thus:

```
s/\\.\*/backslash dot star/
```

changes '\.\*' into 'backslash dot star'.

Since a period means 'any character', the question naturally arises of what to do when you really want a period. For example, how do you convert the line:

```
Now is the time.
```

into:

```
Now is the time?
```

Use the backslash '\' here as well to turn off any special meaning that the next character might have; in particular, '\.' converts the '.' from a 'match anything' into a period, so you can use it to replace the period in 'Now is the time.', type:

```
s/\./?/p
Now is the time?
```

`ed` treats the pair of characters '\.' as a single real period.

You can also use the backslash when searching for lines that contain a special character. Suppose you are looking for a line that contains:

```
.PP
```

The search for `.PP` finds:

```
/.PP/
THE APPLICATION OF ...
```

because the `.` matches the letter `'A'`. But if you say:

```
/\ .PP/
```

you will find only lines that contain `.PP`.

Consider finding a line that contains a backslash. The search:

```
/\ /
```

won't work, because the `'\'` isn't a literal `'\'`, but instead means that the second `'/'` no longer delimits the search. But by preceding a backslash with another one, you can search for a literal backslash. Thus:

```
/\\ /
```

does work. Similarly, you can search for a forward slash `'/'` with:

```
/\/ /
```

The backslash turns off the meaning of the immediately following `'/'` so that it doesn't terminate the `/.../` construction prematurely.

As an exercise, before reading further, find two substitute commands that each convert the line:

```
\x\ . \y
```

into the line:

```
\x\y
```

Here are several solutions; verify that each works as advertised.

```
s/\\\. //
s/x . /x/
s/ . y/y/
```

Here are a couple of miscellaneous notes about backslashes and special characters. First, you can use any character to delimit the pieces of an `s` command: there is nothing sacred about slashes. But you must use slashes for context searching. For instance, in a line that contains a lot of slashes already, like:

```
//exec //sys.fort.go // etc...
```

you could use a colon as the delimiter — to delete all the slashes, type:

```
s/::g
```

When you are adding text with `a` or `i` or `c`, the backslash is not special, and you should only put in one backslash for each one you really want.

### Specifying the End of Line — the Dollar Sign `$`

The dollar-sign, `$`, denotes the end of a line:

```
/string$/
```

only finds an occurrence of 'string' that is at the end of some line. This implies, of course, that:

```
/^string$/
```

finds a line that contains just 'string', and:

```
/^.$/
```

finds a line containing exactly one character.

As an obvious use, suppose you have the line:

```
Now is the
```

and you wish to add the word 'time' to the end. Use the `$` like this:

```
s/$/ time/p  
Now is the time
```

Notice that a space is needed before 'time' in the substitute command, or you will get:

```
Now is thetime
```

As another example, replace the second comma in a line with a period without altering the first comma. Type:

```
s/,,$/./p  
Now is the time, for all good men,
```

The `$` sign here specifies the comma at the end of the sentence. Without it, of course, `s` operates on the first comma to produce:

```
s/,././p  
Now is the time. for all good men,
```

As another example, to convert:

```
Now is the time.
```

into:

```
Now is the time?
```

as you did earlier, you can use:

```
s/.$/?/p  
Now is the time?
```

Like '.', the `$` has multiple meanings depending on context. In the line:

```
$s/$/$/
```

### Specifying the Beginning of the Line — the Circumflex $\wedge$

the first  $\$$  refers to the last line of the file, the second refers to the end of that line, and the third is a literal dollar sign, to be added to that line.

The circumflex  $\wedge$  signifies the beginning of a line. Thus:

```
/^string/
string
```

finds 'string' only if it is at the beginning of a line, but not:

```
the string...
```

You can also use  $\wedge$  to insert something at the beginning of a line. For example, to place a space at the beginning of the current line, type:

```
s/^/ /
```

You can combine metacharacters. To search for a line that contains *only* the characters `.PP` by typing:

```
/^\.PP$/
```

### Matching Anything — the Star $*$

Suppose you have a line that looks like this:

```
text x          y text
```

where *text* stands for lots of text, and there are some indeterminate number of spaces between the 'x' and the 'y'. Suppose the job is to replace all the spaces between 'x' and 'y' by a single space. The line is too long to retype, and there are too many spaces to count. What now?

This is where the metacharacter  $*$  comes in handy. A character followed by a star stands for as many consecutive occurrences of that character as possible. To refer to all the spaces at once, say:

```
s/x *y/x y/
```

The construction  $*$  means 'as many spaces as possible'. Thus `x *y` means 'an x, as many spaces as possible, then a y'.

You can use the star with any character, not just the space. If the original example was instead:

```
text x-----y text
```

then you can replace all `-` signs by a single space with the command:

```
s/x-*y/x y/
```

Finally, suppose that the line was:

```
text x.....y text
```

Can you see what trap lies in wait for the unwary? What will happen if you blindly type:

```
s/x.*y/x y/
```

The answer, naturally, is that it depends. If there are no other x's or y's on the line, then everything works, but it's blind luck, not good management.

Remember that '.' matches *any* single character. Then '\*' matches as many single characters as possible, and unless you're careful, it can eat up a lot more of the line than you expected. If the line was, for example, like this:

```
text x text x.....y text y text
```

then saying:

```
s/x.*y/x y/
```

takes everything from the *first* 'x' to the *last* 'y'. In this example, this is more than you wanted.

The solution, of course, is to turn off the special meaning of '.' with

```
\.:
```

```
s/x\.y/x y/
```

Now everything works, for '\.\*' means 'as many *periods* as possible'.

The dot is useful in conjunction with '\*', a repetition character; 'a\*' is a shorthand for 'any number of 'a' s', so '.\*' matches any number of anythings. Use this like:

```
s/.*/stuff/
```

which changes an entire line, or:

```
s/.*,//
```

which deletes all characters in the line up to and including the last comma. Since '\*' finds the longest possible match, this goes up to the last comma.

There are times when the pattern '.\*' is exactly what you want. For example, use:

```
Now is the time for all good men ....
```

```
s/ for.*/./p
```

```
Now is the time.
```

The '.\*' replaces all of the characters from the space before the word 'for' with a dot. The string 'Now is the time.' is the result in this example.

There are a couple of additional pitfalls associated with '\*' that you should be aware of. First note that 'as many as possible' means *zero* or more. The fact that zero is a legitimate possibility is sometimes rather surprising. For example, if your line contained:

```
text xy text x                y text
```

and you said:

```
s/x *y/x y/
```

the *first* 'xy' matches this pattern, for it consists of an 'x', zero spaces, and a 'y'. The result is that the substitute acts on the first 'xy', and does not touch the later

one that actually contains some intervening spaces.

The way around this, if it matters, is to specify a pattern like:

```
/xb*y/
```

where **b** represents a blank. This describes ‘an x, a space, then as many more spaces as possible, then a y’, in other words, one or more spaces.

The other startling behavior of **\*** is again related to the fact that zero is a legitimate number of occurrences of something followed by a star. The following command does not produce what was intended:

```
abcdef
s/x*/y/g
P
yaybycydyeyfy
```

The reason for this behavior again, is that zero is a legal number of matches, and there are no x’s at the beginning of the line (so that gets converted into a ‘y’), nor between the ‘a’ and the ‘b’ (so that gets converted into a ‘y’), nor ... and so on. Make sure you really want zero matches; if not, in this case write:

```
s/xx*/y/g
```

**xx\*** is ‘one or more ‘x’s’.

## Character Classes — Brackets [ ]

The **[** and **]** brackets form ‘character classes’. Any characters can appear within a character class, and just to confuse the issue, there are essentially no special characters inside the brackets; even the backslash doesn’t have a special meaning. For example, to match any single digit, use:

```
/[0123456789]/
```

Any one of the characters inside the braces will cause a match. It is a nuisance to have to spell out the digits, so you can abbreviate them as **[0–9]**. Similarly, **[a–z]** stands for the lower-case letters, and **[A–Z]** for upper case.

Suppose that you want to delete any numbers that appear at the beginning of all lines of a file. You might first think of trying a series of commands like:

```
1, $s/^1*//
1, $s/^2*//
1, $s/^3*//
```

and so on, but this is clearly going to take forever if the numbers are at all long. Unless you want to repeat the commands over and over until all numbers are gone, you must get all the digits on one pass. This is the purpose of the brackets **[** and **]**.

Another example: To match zero or more digits (an entire number), and to delete all digits from the beginning of all lines, type:

```
1, $s/^[0123456789]*//
```

To search for special characters, for example, you can say:

```
[.\$^[ ]/
```

Within [...], the [ is not special. To get a ] into a character class, make it the first character.

As a final frill on character classes, you can specify a class that means 'none of the following characters'. To do this, begin the class with a caret (^) to stand for 'any character *except* a digit':

```
[^0-9]
```

Thus you might find the first line that does not begin with a tab or space by a search like:

```
/^[^(space)(tab)]/
```

Within a character class, the circumflex has a special meaning only if it occurs at the beginning. Just to convince yourself, verify that to find a line that doesn't begin with a circumflex, you type:

```
/^[^^]/
```

#### 4.6. Cutting and Pasting with the Editor Moving Lines Around

##### Moving Text Around — the Move Command `m`

`ed` has commands for manipulating individual lines or groups of lines in files.

There are several ways to move text around in a file.

Use the *move* command `m` for cutting and pasting — you can move a group of lines from one place to another in the buffer. Suppose you want to put the first three lines of the buffer at the end instead. You could do it by saying:

```
1,3w temp
$r temp
1,3d
```

This is the brute force way; that is, you write the paragraph into a temporary file, read in the temporary file at the end, and then delete it from its current position. As another example, consider:

```
./^\.PP/-w temp
.///-d
$r temp
```

That is, from where you are now ('.') until one line before the next `.PP` (`/^\.PP/-`), write into *temp*. Then delete the same lines. Finally, read in *temp* at the end.

But you can do it a lot easier with `m`, so you can do a whole operation at one crack.

```
1,3m$
```

The general case is:

```
start line, end line m after this line
```

Notice that there is a third line to be specified — the place where the moved stuff

gets put.

If you try:

```
1,5m3
?
```

ed reminds you that you can't do this.

The `m` command is like many other ed commands in that it takes up to two line numbers in front that tell what lines are to be moved. It is also *followed* by a line number that tells where the lines are to go. Thus:

```
line1, line2 m line3
```

says to move all the lines between 'line1' and 'line2' after 'line3'. Naturally, any of 'line1' etc., can be patterns between slashes, dollar signs, or other ways to specify lines.

Of course you can specify the lines to be moved by context searches; if you had:

```
First paragraph
...
end of first paragraph.
Second paragraph
...
end of second paragraph.
```

you could reverse the two paragraphs like this:

```
/Second/,/end of second/m/First/-1
```

Notice the `-1`: the moved text goes *after* the line mentioned. Dot gets set to the last line moved. Suppose you want to move a paragraph from its present position in a paper to the end. How would you do it? As a hint, suppose each paragraph in the paper begins with the formatting command `.PP`. Think about it and write down the details before reading on.

Suppose again that you're sitting at the first line of the paragraph. Then you can say:

```
.,/^\.PP/-m$
```

That's all.

As another example of a frequent operation, you can reverse the order of two adjacent lines by moving the first one after the second. Suppose that you are positioned at the first. Then, to move line dot to one line after line dot, type:

```
m+
```

If you are positioned on the second line, and want to do the reverse, type:

```
m--
```

As you can see, `m` is more succinct and direct than writing, deleting and re-reading. When is brute force better? This is a matter of personal taste — do what you have most confidence in. The main difficulty with `m` is that if you use patterns to specify both the lines you are moving and the target, you have to take

care that you specify them properly, or you may well not move the lines you thought you did. The result of a botched `m` command can be a mess. Doing the job a step at a time makes it easier for you to verify at each step that you accomplished what you wanted to. It's also a good idea to use a `w` command before doing anything complicated; then if you goof, it's easy to back up to where you were.

### Substituting Newlines

You can split a single line into two or more shorter lines by 'substituting in a newline'. As the simplest example, suppose a line has gotten unmanageably long because of editing or merely because it was unwisely typed. If it looks like:

```
text xy text
```

you can break it between the 'x' and the 'y' like this:

```
s/xy/x\  
y/
```

This is actually a single command, although it is typed on two lines. Bearing in mind that '`\`' turns off special meanings, it seems relatively intuitive that a '`\`' at the end of a line would make the newline there no longer special.

You can in fact make a single line into several lines with this same mechanism. As a large example, consider underlining the word 'very' in a long line by splitting 'very' onto a separate line, and preceding it by the `nroff` formatting command `.ul`.

```
text a very big text
```

To convert the line into four shorter lines, preceding the word 'very' by the line `.ul`, and eliminating the spaces around the 'very', all at the same time, type:

```
s/ very /\  
.ul\  
very\  
/
```

When a newline is substituted in, dot is left pointing at the last line created.

### Joining Lines — the Join Command `j`

You may also join lines together, but use the *join* command `j` for this instead of `s`. Given the lines:

```
Now is  
the time
```

and supposing that dot is set to the first of them, then the command:

```
j
```

joins them together. No blanks are added, which is why we carefully showed a blank at the beginning of the second line.

All by itself, a `j` command joins line `dot` to line `dot+1`, but any contiguous set of lines can be joined. Just specify the starting and ending line numbers. For example:

**1,\$jP**

joins all the lines into one big one and displays it.

### Rearranging a Line with \ ( ... \)

Skip this section if this is the first time you're reading this chapter. Recall that `&` stands for whatever was matched by the left side of an `s` command. In much the same way you can capture separate pieces of what was matched; the only difference is that you have to specify on the left side just what pieces you're interested in.

Suppose, for instance, that you have a file of lines that consist of names in the form:

```
Smith, A. B.
Jones, C.
```

and so on, and you want the initials to precede the name, as in:

```
A. B. Smith
C. Jones
```

It is possible to do this with a series of editing commands, but it is tedious and error-prone. (It is instructive to figure out how it is done, though.)

The alternative is to 'tag' the pieces of the pattern, in this case, the last name, and the initials, and then rearrange the pieces. On the left side of a substitution, if part of the pattern is enclosed between `\(` and `\)`, whatever matched that part is remembered, and available for use on the right side. On the right side, the symbol `\1` refers to whatever matched the first `\(...\)` pair, `\2` to the second `\(...\)`, and so on.

The command:

```
1,$s/^\([^,]*\) , *\(.*\)/\2 \1/
```

although hard to read, does the job. The first `\(...\)` matches the last name, which is any string up to the comma; this is referred to on the right side with `\1`. The second `\(...\)` is whatever follows the comma and any spaces, and is referred to as '`\2`'.

Of course, with any editing sequence this complicated, it's foolhardy to simply run it and hope. The global commands `g` and `v` provide a way for you to display exactly those lines which were affected by the substitute command, and thus verify that it did what you wanted in all cases.

### Marking a Line — the Mark Command `k`

You can mark a line with a particular name so you can refer to it later by name, regardless of its actual line number. This can be handy for moving lines, and for keeping track of them as they move. The *mark* command is `k`. To mark the current line with the name `x`, use:

```
kx
```

If a line number precedes the `k`, that line is marked. The mark name must be a single lower-case letter. Now you can refer to the marked line with the address:

'x

Marks are most useful for moving things around. Find the first line of the block to be moved, and mark it with 'a. Then find the last line and mark it with 'b. Now position yourself at the place where the stuff is to go and say:

'a, 'bm.

Bear in mind that only one line can have a particular mark name associated with it at any given time.

#### Copying Lines — the Transfer Command t

We mentioned earlier the idea of saving a line that was hard to type or used often, to cut down on typing time. Of course this can be more than one line, in which case the saving is presumably even greater.

ed provides another command, called t (*transfer*) for making a copy of a group of one or more lines at any point. This is often easier than writing and reading.

The t command is identical to m, except that instead of moving lines, it simply duplicates them at the place you named. Thus, to duplicate the entire contents that you are editing, use:

1, \$t\$

A more common use for t is for creating a series of lines that differ only slightly. For example, you can say:

```
a
..... x ..... (long line)
.
t.          (make a copy)
s/x/y/     (change it a bit)
t.          (make third copy)
s/y/z/     (change it a bit)
```

and so on.

#### 4.7. Escaping to the Shell with !

Sometimes it is convenient to be able to temporarily escape from the editor to use some Shell command without leaving the editor. Use the ! (escape) command to do this.

To suspend your current editing state and execute the shell command you asked for, type:

```
!any shell command
!
```

When the command finishes, ed will signal you by displaying another !; at that point, you can resume editing.

You can really do *any* shell command, including another ed. This is quite common, in fact. In this case, you can even do another !.

## 4.8. Supporting Tools

There are several tools and techniques that go along with the editor, all of which are relatively easy once you know how `ed` works, because they are all based on the editor. This section gives some fairly cursory examples of these tools, more to indicate their existence than to provide a complete tutorial. For more information on each, refer to the *Commands Reference Manual for the Sun Workstation*.

### Editing Scripts

If you have a fairly complicated set of editing operations to do on a whole set of files, the easiest thing to do is to make up a ‘script’, that is, a file that contains the operations you want to perform, and then apply this script to each file in turn.

For example, suppose you want to change every ‘SUN’ to ‘Sun’ and every ‘SYSTEM’ to ‘System’ in a large number of files. Then put into a file, which we’ll call *changes*, the lines:

```
g/SUN/s//Sun/g
g/SYSTEM/s//System/g
w
q
```

Now you can say:

```
hostname% ed file1 <script
hostname% ed file2 <script
...
```

This causes `ed` to take its commands from the prepared script called *changes*. Notice that you have to plan the whole job in advance.

And of course by using the Sun UNIX command interpreter, the shell, you can cycle through a set of files automatically, with varying degrees of ease.

### Matching Patterns with `grep`

Sometimes you want to find all occurrences of some word or pattern in a set of files, to edit them or perhaps just to verify their presence or absence. You can edit each file separately and look for the pattern of interest, but if there are many files, this can get very tedious, and if the files are really big, it may be impossible because of limits in `ed`.

The program `grep` gets around these limitations. The search patterns that are described in this chapter are often called ‘regular expressions’, and ‘`grep`’ stands for ‘general regular expression, print.’ That describes exactly what `grep` does — it displays every line in a set of files that contains a particular pattern. Thus, to find ‘thing’ wherever it occurs in any of the files *file1*, *file2*, etc., type:

```
hostname% grep 'thing' file1 file2 file3 ...
hostname%
```

`grep` also indicates the file in which the line was found, so you can later edit it if you like.

The pattern represented by ‘thing’ can be any pattern you can use in the editor, since `grep` and `ed` use exactly the same mechanism for pattern searching. It is wisest always to enclose the pattern in the single quotes ‘...’ if it contains any non-alphabetic characters, since many such characters also mean something special to the Sun UNIX command interpreter, the shell. If you don’t quote them, the command interpreter will try to interpret them before `grep` gets a chance.

There is also a way to find lines that *do not* contain a pattern:

```
hostname% grep -v 'thing' file1 file2 ...
hostname%
```

finds all lines that don't contain 'thing'. The `-v` must occur in the position shown. Given `grep` and `grep -v`, it is possible to do things like selecting all lines that contain some combination of patterns. For example, to get all lines that contain 'x' but not 'y', use:

```
hostname% grep x file... | grep -v y
hostname%
```

The notation `|` is a 'pipe', which causes the output of the first command to be used as input to the second command; see the *Beginner's Guide to the Sun Workstation* for an introduction to 'piping.' See the *Commands Reference Manual for the Sun Workstation* for details on `grep`.

#### 4.9. Summary of Commands and Line Numbers

The general form of `ed` commands is the command name, perhaps preceded by one or two line numbers, and, in the case of `e`, `r`, and `w`, followed by a filename. Only one command is allowed per line, but a `p` command may follow any other command, except for `e`, `r`, `w` and `q`.

- a Append, that is, add lines to the buffer at line dot, unless a different line is specified. Type a '.' on a new line to terminate appending. Dot is set to the last line appended.
- c Change the specified lines to the new text that follows. Type a '.' as with a to terminate the change. If no lines are specified, replace line dot. Dot is set to last line changed.
- d Delete the lines specified. If none is specified, delete line dot. Dot is set to the first undeleted line, unless '\$' is deleted, in which case dot is set to '\$'.
- e Edit new file. Any previous contents of the buffer are thrown away, so use a w beforehand.
- f Print remembered filename. If a name follows f the remembered name will be set to it.
- g The command:  

```
g/--/commands
```

executes the commands on those lines that contain '---', which can be any context search expression.
- i Insert lines before specified line (or dot) until a '.' is typed on a new line. Dot is set to last line inserted.
- m Move lines specified to after the line named after m. Dot is set to the last line moved.
- p Display specified lines. If none is specified, display line dot. A single line number is equivalent to *line-number* p. Type a single

- RETURN to show `.+1`, the next line.
- q** Quit `ed`. This wipes out all text in buffer if you give it twice in a row without first giving a `w` command.
- r** Read a file into the buffer at the end unless an address is specified. Dot is set to the last line read.
- s** The command:  
**`s/string1/string2/`**  
 substitutes the characters 'string2' into 'string1' in the specified lines. If no lines are specified, make the substitution in line dot. Dot is set to last line in which a substitution took place, which means that if no substitution took place, dot is not changed. An `s` changes only the first occurrence of 'string1' on a line; to change all of them, type a `g` after the final slash.
- v** The command:  
**`▼/---/commands`**  
 executes *commands* on those lines that *do not* contain '---'.
- w** Write out buffer into a file. Dot is not changed.
- .=** Show value of dot (current line number). An '=' by itself shows the value of '\$.' (number of the last line in the buffer).
- !** The line:  
**`!command`**  
 executes *command* as a Sun UNIX shell command.
- /-----/** Context search. Search for next line that contains this string of characters and display it. Dot is set to the line where string was found. Search starts at `+.1`, wraps around from '\$' to 1, and continues to dot, if necessary.
- ?-----?** Context search in reverse direction. Start search at `.-1`, scan to 1, wrap around to '\$.'



**Section II:**  
**Viewing Files**



Using view



**NAME**

**view** – view a file without changing it using the *vi* visual editor

**SYNOPSIS**

**view** [**-t** tag] [**-r**] [**+command**] [**-l**] [**-wn**] name ...

**DESCRIPTION**

*View* uses the *vi* (visual) or display oriented text editor to browse through a file interactively without actually making any changes to the file. It is possible to get to the command mode of *ex* from within *view* and vice-versa, just as when using *vi*.

**FILES**

See *ex*(1).

**SEE ALSO**

*ex* (1), *edit* (1), *vi*(1), “Vi Quick Reference” card,  
*Using vi, the Visual Display Editor in  
Editing and Text Processing on the Sun Workstation.*



## Using cat



**NAME**

**cat** – concatenate and display

**SYNOPSIS**

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

**DESCRIPTION**

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

**% cat goodies**

displays the contents of *goodies* on the standard output, and

**% cat file1 file2 >file3**

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

If no filename argument is given, or if the argument ‘-’ is given, *cat* reads from the standard input file. If the standard input is a terminal, input is terminated by a ^D.

**OPTIONS**

- u** makes the output completely unbuffered. If **-u** is not used, output is buffered in 1024-byte blocks, or line-buffered if standard output is a terminal.
- n** precedes each line output with its line number.
- b** numbers the lines, as **-n**, but omits the line numbers from blank lines.
- s** substitutes a single blank line for multiple adjacent blank lines.
- v** displays non-printing characters so that they are visible. Control characters print like ^X for control-x; the delete character (octal 0177) prints as ^?. Non-ASCII characters (with the high bit set) are displayed as M- (for meta) followed by the character of the low 7 bits.
- e** displays non-printing characters, as **-v**, and in addition displays a ‘\$’ character at the end of each line.
- t** displays non-printing characters, as **-v**, and in addition displays tab characters as ‘^I’.

**SEE ALSO**

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

**BUGS**

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



# Using head

1

1

1

**NAME**

**head** – display first few lines of specified files

**SYNOPSIS**

**head** [ **-count** ] [ **file ...** ]

**DESCRIPTION**

*Head* copies the first *count* lines of the specified file(s), or of the standard input if no filename is given, to the standard output. The default value of *count* is 10 lines.

When more than one file is specified, *head* places a marker at the start of each file which looks like:

```
==> filename <==
```

Thus, a common way to display a set of short files, identifying each one, is:

```
gaia% head -9999 file1 file2 ...
```

**EXAMPLE**

```
gaia% head -4 /usr/man/man1/{cat,head,tail}.1
```

```
==> /usr/man/man1/cat.1 <==
```

```
.TH CAT 1 "2 June 1983"
```

```
.SH NAME
```

```
cat – concatenate and display
```

```
.SH SYNOPSIS
```

```
==> /usr/man/man1/head.1 <==
```

```
.TH HEAD 1 "24 August 1983"
```

```
.SH NAME
```

```
head – display first few lines of specified files
```

```
.SH SYNOPSIS
```

```
==> /usr/man/man1/tail.1 <==
```

```
.TH TAIL 1 "27 April 1983"
```

```
.SH NAME
```

```
tail – display the last part of a file
```

```
.SH SYNOPSIS
```

**SEE ALSO**

more(1), tail(1), cat(1)



# Using tail



**NAME**

**tail** – display the last part of a file

**SYNOPSIS**

**tail** [ *±number* [ *lbc* ] [ *fr* ] ] [ *file* ]

**DESCRIPTION**

*Tail* copies the named *file* to the standard output beginning at a designated place. If no file is named, the standard input is used.

**OPTIONS**

Options are all jammed together, not specified separately with their own **–** signs.

**+number**

Begin copying at distance *+number* from the beginning of the file. *Number* is counted in units of lines, blocks or characters, according to the appended option **l**, **b**, or **c**. When no units are specified, counting is by lines.

**–number**

Begin copying at distance *–number* from the end of the file. *Number* is counted in units of lines, blocks or characters, according to the appended option **l**, **b**, or **c**. When no units are specified, counting is by lines.

**r** Copy lines from the end of the file in reverse order. The default for **r** is to print the entire file this way.

**f** Follow the file as it grows, that is, don't quit at end of file, but rather wait and try to read repeatedly in hopes that the file will grow.

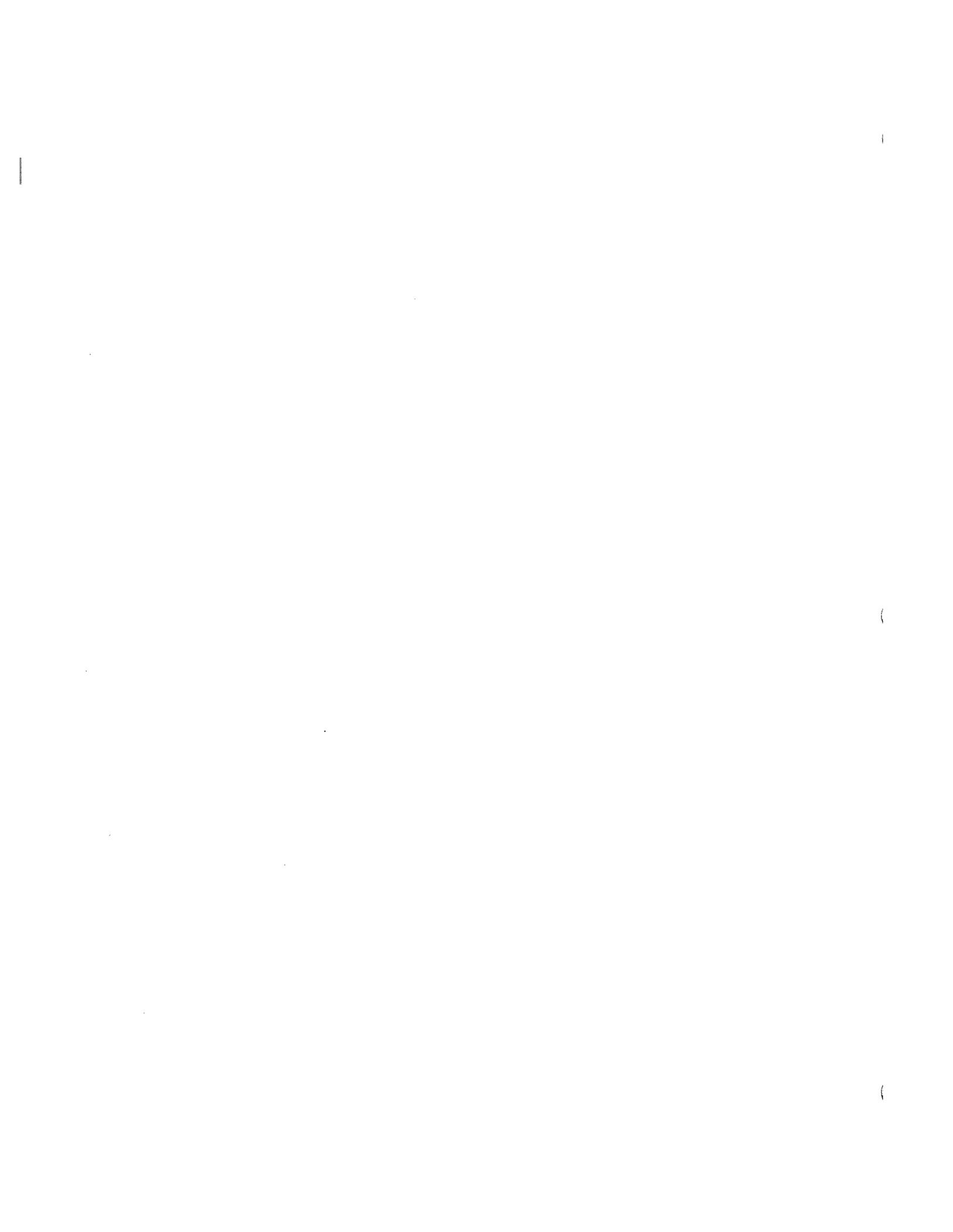
**SEE ALSO**

**dd(1)**

**BUGS**

Tails relative to the end of the file are treasured up in a buffer, and thus are limited in length.

Various kinds of anomalous behavior may happen with character special files.



Using more



## NAME

*more*, *page* – browse through a text file

## SYNOPSIS

**more** [ *-cdfisu* ] [ *-lines* ] [ *+linenumber* ] [ *+/pattern* ] [ *name ...* ]

**page** [ *-cdfisu* ] [ *-lines* ] [ *+linenumber* ] [ *+/pattern* ] [ *name ...* ]

## DESCRIPTION

*More* is a filter which displays the contents of a text file one screenful at a time on a video terminal. It normally pauses after each screenful, and prints ‘--More--’ at the bottom of the screen. *More* displays another line if you type a carriage-return; *more* displays another screenful if you type a space.

If you use the *page* command instead of the *more* command, the screen is cleared before each screenful is displayed (but only if a full screenful is being displayed), and  $k - 1$  rather than  $k - 2$  lines are displayed in each screenful, where  $k$  is the number of lines the terminal can display.

*More* looks in the file */etc/termcap* to determine terminal characteristics, and to determine the default window size. On a terminal capable of displaying 24 lines, the default window size is 22 lines.

*More* looks in the environment variable *MORE* to pre-set any flags desired. For example, if you prefer to view files using the *-c* mode of operation, the *cs*h command “*setenv MORE -c*” or the *sh* command sequence “*MORE='-c' ; export MORE*” would cause all invocations of *more*, including invocations by programs such as *man* to use this mode. Normally, the user will place the command sequence which sets up the *MORE* environment variable in the *.login* or *.profile* file.

If *more* is reading from a file, rather than a pipe, a percentage is displayed along with the --More-- prompt. This gives the fraction of the file (in characters, not lines) that has been read so far.

Other sequences which may be typed when *more* pauses, and their effects, are as follows (*i* is an optional integer argument, defaulting to 1) :

- i* <space> display *i* more lines, (or another screenful if no argument is given)
- ^D display 11 more lines (a “scroll”). If *i* is given, the scroll size is set to *i*.
- d same as ^D (control-D)
- iz* same as typing a space except that *i*, if present, becomes the new window size.
- is* skip *i* lines and print a screenful of lines
- if* skip *i* screenfuls and print a screenful of lines
- q or Q Exit from *more*.
- = Display the current line number.
- v Start up the editor *vi* at the current line.
- h Help command; give a description of all the *more* commands.
- i*/*expr* search for the *i*-th occurrence of the regular expression *expr*. If there are less than *i* occurrences of *expr*, and the input is a file (rather than a pipe), the position in the file remains unchanged. Otherwise, a screenful is displayed, starting two lines before the place where the expression was found, or the end of the pipe, whichever comes first. The user’s erase and kill characters may be used to edit the regular expression. Erasing back past the first column cancels the search command.
- in* search for the *i*-th occurrence of the last regular expression entered.
- ' (single quote) Go to the point from which the last search started. If no search has been performed in the current file, this command goes back to the beginning of the file.

**!command**

invoke a shell with *command*. The characters '%' and '!' in "command" are replaced with the current file name and the previous shell command respectively. If there is no current file name, '%' is not expanded. The sequences "\%" and "\!" are replaced by "%" and "!" respectively.

- i:n** skip to the *i*-th next file given in the command line (skips to last file if *n* doesn't make sense)
- i:p** skip to the *i*-th previous file given in the command line. If this command is given in the middle of printing out a file, *more* goes back to the beginning of the file. If *i* doesn't make sense, *more* skips back to the first file. If *more* is not reading from a file, the bell is rung and nothing else happens.
- :f** display the current file name and line number.
- :q or :Q** exit from *more* (same as q or Q).
- .** (dot) repeat the previous command.

The commands take effect immediately; it is not necessary to type a carriage return. Up to the time when the command character itself is given, the user may type the line kill character to cancel the numerical argument being formed. In addition, the user may type the erase character to redisplay the --More--(xx%) message.

At any time when output is being sent to the terminal, the user can type the quit key (normally control-^). *More* stops sending output, and displays the usual --More-- prompt. The user may then enter one of the above commands in the normal manner. Unfortunately, some output is lost when this is done, due to the fact that any characters waiting in the terminal's output queue are flushed when the quit signal occurs.

*More* sets the terminal to *noecho* mode so that the output can be continuous. Thus what you type does not show on your terminal, except for the / and ! commands.

If the standard output is not a terminal, *more* acts just like *cat*, except that a header is printed before each file in a series.

**OPTIONS**

- lines** Set the size of the window to *lines* lines long instead of the default.
- c** Display each page by redrawing the screen instead of scrolling. This makes it easier to read text while *more* is writing. This option is ignored if the terminal does not have the ability to clear to the end of a line.
- d** Display the message 'Hit space to continue, Rubout to abort' at the end of each screenful. This is useful if *more* is being used as a filter in some setting, such as a class, where users are unsophisticated.
- f** Count logical rather than screen lines. That is, long lines are not folded. This option is recommended if *nroff* output is being piped through *ul*, since the latter may generate escape sequences. These escape sequences contain characters which would ordinarily occupy screen positions, but which do not print when they are sent to the terminal as part of an escape sequence. Thus *more* may think that lines are longer than they actually are, and fold lines erroneously.
- l** Do not treat ^L (form feed) specially. If **-l** is not used, *more* pauses after any line that contains a ^L, as if the end of a screenful had been reached. Also, if a file begins with a form feed, the screen is cleared before the file is printed.
- s** Squeeze multiple blank lines from the output, and replace them with single blank lines. Especially helpful when viewing *nroff* output, this option maximizes the useful information present on the screen.
- u** Normally, *more* handles underlining such as produced by *nroff* in a manner appropriate to the particular terminal: if the terminal can perform underlining or has a stand-out mode, *more* outputs appropriate escape sequences to enable underlining or stand-out mode for underlined information in the source file. The **-u** option suppresses this processing.

**+linenumber**

Start up at *linenumber*.

*+/pattern*

Start up two lines before the line containing the regular expression *pattern*.

#### EXAMPLES

A sample usage of *more* in previewing *nroff* output would be

```
nroff -ms +2 doc.n | more -s
```

#### FILES

*/etc/termcap* Terminal data base

*/usr/lib/more.help* Help file

#### SEE ALSO

*csh(1)*, *man(1)*, *script(1)*, *sh(1)*, *environ(5)*, *termcap(5)*



---

## Revision History

Revision	Date	Comments
A	17 February 1986	First release of Editing Text Files on the Sun Workstation.

---

## Notes

## Vi Quick Reference

### Entering and Leaving vi

% vi <i>name</i>	edit <i>name</i> at top
% vi + <i>n name</i>	... at line <i>n</i>
% vi + <i>name</i>	... at end
% vi -r	list saved files
% vi -r <i>name</i>	recover file <i>name</i>
% vi <i>name</i> ...	edit first; rest via :n
% vi -t <i>tag</i>	start at <i>tag</i>
% vi +/pat <i>name</i>	search for <i>pat</i>
% view <i>name</i>	read only mode
ZZ	exit from vi, saving changes
CTRL-Z	stop vi for later resumption

### The Display

Last line	Error messages, echoing input to : / ? and !, feedback about i/o and large changes.
@ lines	On screen only, not in file.
~ lines	Lines past end of file.
CTRL-x	Control characters, DEL is delete.
tabs	Expand to spaces, cursor at last.

### Vi Modes

Command	Normal and initial state. Others return here. ESC (escape) cancels partial command.
Insert	Entered by a i A I o O c C s S R. Arbitrary text then terminates with ESC character, or abnormally with interrupt.
Last line	Reading input for : / ? or !; terminate with ESC or CR to execute, interrupt to cancel.

### Counts Before vi Commands

line/column number	z G
scroll amount	CTRL-D CTRL-U
replicate insert	a i A I
repeat effect	most rest

### Simple Commands

dw	delete a word
de	... leaving punctuation
dd	delete a line
3dd	... 3 lines
itextESC	insert text <i>abc</i>
cwnewESC	change word to <i>new</i>
easESC	pluralize word
xp	transpose characters

### Interrupting, Cancelling

ESC	end insert or incomplete cmd
CTRL-C	interrupt (or DEL)
CTRL-L	refresh screen if scrambled

### File Manipulation

:w	write back changes
:wq	write and quit
:q	quit
:q!	quit, discard changes
:e <i>name</i>	edit file <i>name</i>
:e!	reedit, discard changes
:e + <i>name</i>	edit, starting at end
:e + <i>n</i>	edit starting at line <i>n</i>
:e #	edit alternate file
CTRL-^	synonym for :e #
:w <i>name</i>	write file <i>name</i>
:w! <i>name</i>	overwrite file <i>name</i>
:sh	run shell, then return
!: <i>cmd</i>	run <i>cmd</i> , then return
:n	edit next file in arglist
:n <i>args</i>	specify new arglist
:f	show current file and line
CTRL-G	synonym for :f
:ta <i>tag</i>	to tag file entry <i>tag</i>
CTRL-]	:ta, following word is <i>tag</i>

### Positioning within File

CTRL-F	forward screenfull
CTRL-B	backward screenfull
CTRL-D	scroll down half screen
CTRL-U	scroll up half screen
G	goto line (end default)
/pat	next line matching <i>pat</i>
?pat	prev line matching <i>pat</i>
n	repeat last / or ?
N	reverse last / or ?
/pat/+n	n'th line after <i>pat</i>
?pat?~n	n'th line before <i>pat</i>
]]	next section/function
[[	previous section/function
%	find matching ( ) { or }

### Adjusting the Screen

CTRL-L	clear and redraw
CTRL-R	retype, eliminate @ lines
zCR	redraw, current at window top
z-	... at bottom
z.	... at center
/pat/z-	<i>pat</i> line at bottom
zn.	use <i>n</i> line window
CTRL-E	scroll window down 1 line
CTRL-Y	scroll window up 1 line

**Marking and Returning**

``	previous context
''	... at first non-white in line
mx	mark position with letter <i>x</i>
`x	to mark <i>x</i>
ˆx	... at first non-white in line

**Line Positioning**

H	home window line
L	last window line
M	middle window line
+	next line, at first non-white
-	previous line, at first non-white
CR	return, same as +
↓ or j	next line, same column
↑ or k	previous line, same column

**Character Positioning**

^	first non-blank
0	beginning of line
\$	end of line
h or →	forward
l or ←	backwards
CTRL-H	same as ←
space	same as →
fx	find <i>x</i> forward
Fx	<i>f</i> backward
tx	upto <i>x</i> forward
Tx	back upto <i>x</i>
;	repeat last <i>f F t</i> or <i>T</i>
,	inverse of ;
	to specified column
%	find matching ( { ) or }

**Words, Sentences, Paragraphs**

w	word forward
b	back word
e	end of word
)	to next sentence
}	to next paragraph
(	back sentence
{	back paragraph
W	blank delimited word
B	back W
E	to end of W

**Commands for LISP**

)	Forward s-expression
}	... but don't stop at atoms
(	Back s-expression
{	... but don't stop at atoms

**Corrections During Insert**

CTRL-H	erase last character
CTRL-W	erases last word
erase	your erase, same as CTRL-H
kill	your kill, erase input this line
\	escapes CTRL-H, your erase and kill
ESC	ends insertion, back to command
CTRL-C	interrupt, terminates insert
CTRL-D	backtab over <i>autoindent</i>
CTRL-^D	kill <i>autoindent</i> , save for next
0CTRL-D	... but at margin next also
CTRL-V	quote non-printing character

**Insert and Replace**

a	append after cursor
i	insert before
A	append at end of line
I	insert before first non-blank
o	open line below
O	open above
rx	replace single char with <i>x</i>
R	replace characters

**Operators (double to affect lines)**

d	delete
c	change
<	left shift
>	right shift
!	filter through command
=	indent for LISP
y	yank lines to buffer

**Miscellaneous Operations**

C	change rest of line
D	delete rest of line
s	substitute chars
S	substitute lines
J	join lines
x	delete characters
X	... before cursor
Y	yank lines

**Yank and Put**

p	put back lines
P	put before
"xp	put from buffer <i>x</i>
"xy	yank to buffer <i>x</i>
"xd	delete into buffer <i>x</i>

**Undo, Redo, Retrieve**

u	undo last change
U	restore current line
.	repeat last change
"dp	retrieve <i>d</i> 'th last delete

## Ex Quick Reference

### Entering/Leaving ex

% ex <i>name</i>	edit <i>name</i> , start at end
% ex + <i>n name</i>	... at line <i>n</i>
% ex - <i>t tag</i>	start at <i>tag</i>
% ex -r	list saved files
% ex -r <i>name</i>	recover file <i>name</i>
% ex <i>name</i> ...	edit first; rest via : <i>n</i>
% ex -R <i>name</i>	read only mode
: x	exit, saving changes
: q!	exit, discarding changes

### ex States

Command	Normal and initial state. Input prompted for by :. Your kill character cancels partial command.
Insert	Entered by a i and c. Arbitrary text then terminates with line having only . character on it or abnormally with interrupt.
Open/visual	Entered by open or vi, terminates with Q or ^.

### ex Commands

abbrev	ab	next	n	unabbrev	una
append	a	number	nu	undo	u
args	ar	open	o	unmap	unm
change	c	preserve	pre	version	ve
copy	co	print	p	visual	vi
delete	d	put	pu	write	w
edit	e	quit	q	xit	x
file	f	read	re	yank	ya
global	g	recover	rec	window	z
insert	i	rewind	rew	escape	!
join	j	set	se	shift	<
list	l	shell	sh	print next	CR
map		source	so	resubst	&
mark	ma	stop	st	rshift	>
move	m	substitute	s	scroll	^D

### ex Command Addresses

<i>n</i>	line <i>n</i>	/ <i>pat</i>	next with <i>pat</i>
.	current	? <i>pat</i>	previous with <i>pat</i>
\$	last	<i>x-n</i>	<i>n</i> before <i>x</i>
+	next	<i>x,y</i>	<i>x</i> through <i>y</i>
-	previous	^ <i>x</i>	marked with <i>x</i>
+ <i>n</i>	<i>n</i> forward	''	previous context
%	1,\$		

### Specifying Terminal Type

% setenv TERM <i>type</i>	(for <i>csh</i> )
\$ TERM= <i>type</i> ; export TERM	(for <i>sh</i> )

See also *tset* in the user's manual.

### Some Terminal Types

2621	43	adm31	dw1	h19
2645	733	adm3a	dw2	i100
300s	745	c100	gt40	mime
33	act4	dm1520	gt42	owl
37	act5	dm2500	h1500	t1061
4014	adm3	dm3025	h1510	vt52

### Initializing Options

EXINIT	place set's here in environment var.
set x	enable option
set nox	disable option
set x= <i>val</i>	give value <i>val</i>
set	show changed options
set all	show all options
set x?	show value of option x

### Useful Options

autoindent	ai	supply indent
autowrite	aw	write before changing files
ignorecase	ic	in scanning
llsp		( ) { } are s-exp's
llst		print ^I for tab, \$ at end
magic		. [ * special in patterns
number	nu	number lines
paragraphs	para	macro names which start ...
redraw		simulate smart terminal
scroll		command mode lines
sections	sect	macro names ...
shiftwidth	sw	for < >, and input ^D
showmatch	sm	to ) and } as typed
slowopen	slow	choke updates during insert
window		visual mode lines
wrapscan	ws	around end of buffer
wrapmargin	wm	automatic line splitting

### Scanning Pattern Formation

^	beginning of line
\$	end of line
.	any character
\<	beginning of word
\>	end of word
[ <i>str</i> ]	any char in <i>str</i>
[^ <i>str</i> ]	... not in <i>str</i>
[ <i>x-y</i> ]	... between <i>x</i> and <i>y</i>
*	any number of preceding

