Text Editors and Processors HP-UX Concepts and Tutorials

HP Part Number 97089-90022

Hewlett-Packard Company
3404 East Harmony Road, Fort Collins, Colorado 80525

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

WARRANTY

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

August 1986...Edition 1

October 1987...Edition 2. Replaced *vi* and *ex* tutorials with new *vi/ex* tutorial and replaced *sed* tutorial with new material. Added new Introduction section and a short section on Regular Expressions.

Preface

A Note from the Author

This Second Edition of the *Text Editors and Processors* volume of *HP-UX Concepts and Tutorials* represents a significant departure from its predecessor. This version contains two brand new, major tutorials for the *vi/ex* editor and the *sed* editor as well as new sections that provide a short overview of available editors and where each is most commonly used plus a short introduction to regular expressions. The new editor tutorials were developed from a broad base of experience: nearly three years of using *vi* daily, including several months of exploring the obscure corners of both *vi* and *sed* to determine why they have presented so much difficulty to many who have attempted to learn them well.

The result of this effort is two new tutorials that are written from the point of view that a tutorial should, like a tutor, teach you how to start from essentially nothing and, without further assistance from another human being, become reasonably expert in using the program described. Whether you are a secretary who just received the announcement that you have to learn to do your word processing on an HP-UX system, an engineer who has to write reports and accomplish other documentation tasks, or an experienced programmer who is interested in some advanced techniques, these tutorials provide much enlightenment as well as some useful tricks of the trade.

Until now, there has been a painful lack of quality and thorough books available that describe these two powerful and highly productive tools. Frankly, many users, especially newcomers, find these editors either terribly intimidating or so difficult to understand that they are unwilling to use them unless forced to do so. These new tutorials are designed so that you can start as a novice with the first few chapters or first few pages, and start doing a few simple but useful things, then, as you become more comfortable with the simple aspects of each editor, expand your knowledge into more advanced or specialized areas until, with sufficient commitment to study and learning, you can become highly expert in the use of either program.

To create documents of this quality requires months of sometimes painful effort to explore and discover many of the hidden features that the original software authors built in, but either didn't bother to document, or their documentation has been lost to most users. The *vi*/*ex* tutorial, for example is well over twice as large as originally intended because, after using the editor for over two years, the author of the tutorial suddenly discovered a wealth of obscure but very useful commands and features that had previously been largely unknown and undocumented. After investing many months into thorough exploration and testing of the examples, commands, and capabilities covered in these tutorials, they are believed to be two of the most, if not the most complete documents available for the *vi*, *ex*, and *sed* editors. The *sed* tutorial was written and expanded after extensive study of various documents, especially the validation test software used to verify the correct operation of the editor program itself as well as SVID compatibility before HP-UX is shipped to customers. These resources were very valuable in forcing *sed* to reveal many obscure capabilities.

Our desire to provide new information as soon as possible has led to the release of these tutorials before any work could be done on other tutorials we would like to improve, and before the section on regular expressions could be fully developed. However, we are sure the quality of what is now available will be sufficient to obtain your forgiveness for the imperfect state of those other areas. The enthusiastic acceptance of the new tutorials by those who have had an opportunity to review and use preliminary drafts indicate that you should find their pages quite valuable as a learning tool.

Learning Suggestions

It has been said that:

People who think learning is not fun don't know much about either.

The new additions to this document are intended to replace the friend looking over your shoulder, watching your every move and pointing out easier ways (in fact, the introductory part of the *vi* tutorial is derived closely from an afternoon of one-on-one teaching a secretary with hardly any HP-UX experience to use the editor for some simple, but still somewhat tricky text reformatting). However, since printed pages cannot observe, it is up to you, the user, to exert some effort in gleaning from the words they contain the many opportunities that these very useful and not unfriendly (though they may seem otherwise at times) text processing tools provide for you to get more done in less time with less effort. Many of the examples are the direct result of users asking "How can I...?".

The new words contained herein have been scrutinized by many users, both experienced, and inexperienced. It has been found that few of their questions remain unanswered. Special thanks to Ray Liere of Vantage Consulting and Research Corporation as well as many reviewers within HP are in order. Their comments have been most helpful in making it possible to provide top quality learning materials.

Some users of this manual, like a farmer who plants in the Spring to reap in the Autumn, have found that effort invested in the labor of learning yields a generous reward in making programming, writing, and other text-handling tasks much easier. On the other hand, others who restricted their learning to the obvious "easy stuff", found themselves, like the carpenter who used a rock for a hammer, spending more time in under-productive effort than it would have taken to learn easier techniques. We have tried to make the task as enjoyable as possible. The choice is yours. The tools are in your hands.

We would appreciate any comments you may have about the usefulness of this manual or other topics you would like to see covered. Please feel free to write or use the comment card in the back of this volume.

HP-UX Documentation Staff
Hewlett-Packard Company
Fort Collins Systems Division
3404 East Harmony Road
Fort Collins, CO 80525

Table of Contents

Introduction to Text Editors and Processors

1

HP-UX is a very large operating system, and to explain it fully would require a horrible number of manuals. Since you are not interested in burying yourself in a library for the next year or so to learn it, we have created this manual so you can use the powerful capabilities of HP-UX to solve your text editing and processing problems without investing a lot of effort learning about what you don't really need or want to know. Of course, as you gain experience, you will want to use more sophisticated techniques. Advanced topics are also covered, but only after clearly explaining the simple things first.

HP-UX can be likened to a large international airport. As you enter the terminal, there are hundreds of planes outside, each ready to take you to a different destination. When you log in on an HP-UX system, you have hundreds of commands at your disposal to perform almost any task you can conceive. However, you probably have a clearly defined result you want to obtain, and don't need to use more than a very small percentage of the available commands. Much like an airport guide, this manual will lead you down the various corridors and concourses toward your desired destination. If you prefer, you can ignore the many distractions that lie on every side.

Where Do I Go from Here?

This manual falls in the category of *Almost Everything You Will Ever Need to Know About HP-UX Text Processing* (*but didn't understand well enough to know what to ask*). It discusses and explains how to use the following text editors and processing programs:

- *vi*: An interactive editor that maintains a continually updated display, and its variants *view* and *vedit*. This is the most popular HP-UX editor and is well-suited to a broad range of users who use a CRT display and keyboard to interact with HP-UX.
- *ex*: This is the same interactive editor program as *vi*, but it does not use the interactive full-screen visual CRT display. *ex* is intended for use with electro-mechanical teletypewriter terminals, but can also be used with CRT displays as a simple line editor. *ex* has a few useful capabilities that are not accessible from *vi*, but it is easy to switch between the two personalities during a single session to gain access to those capabilities. A variant of *ex* masquerades under the name *edit*, and is documented in a separate tutorial later in this volume.
- The non-interactive streaming editor: *sed*. This program is very helpful when a well-structured set of changes must be made automatically under program control or must be performed on a large number of files (or both). *sed* is a very fast editor and quite flexible when making complex changes in one or more files in a non-interactive environment.
- The line editor: *ed* is most commonly used in shell programs.

A short tutorial on the *awk* text processing program is also included.

Each tutorial topic is identified by a tab divider at the beginning of the section, and explains the operation of a particular editing program. This, the first section provides an overview of text editing and processing in general.

Topics are arranged in sections as follows. Each section may contain several chapters, depending on the nature of the material being presented.

1. The first section in this volume provides a general overview of the HP-UX system and its directory and file structure. General topics such as file structure and how to choose which editor program to use are the primary focus of this section.
2. The second section describes how to use *regular expressions*, a method of defining arbitrary text patterns for locating various text patterns or sequences in files being processed. This information is of limited interest to beginners, but is essential for intermediate and advanced users.

3. The third section demonstrates how to use the *vi* and *ex* editors. *vi* is one of the most popular HP-UX editors. Easy to use and very interactive, it provides you with a comfortable interaction with the file you are altering.
4. The next section describes the *sed* streaming editor that is so useful when performing simple or complex edits on files when manual editing with an interactive editor is too repetitive or time consuming.
5. The *edit* editor, a simplified version of *ex* is of limited interest, but included here for those few who prefer to use it.
6. The *ed* editor, used mainly with shell scripts and programs is documented for those who need to use it or interpret an *ed* script written by someone else.
7. *awk*, a text processing and analyzing program whose name is derived from an acronym composed of the first initial of the last names of its three authors, is described in this volume because of its close relationship to the other materials presented here.

This section of this manual explains briefly what editors are provided with the HP-UX operating system, and where each one is typically used. This should make a newcomer's task of selecting an editor much easier.

Later sections explain each of the different editors on HP-UX, what each is useful for and the kinds of tasks each is well suited to.

After the sections on text editing, various text processing and related topics are explained. At present, only the powerful text processor *awk* which can scan text for patterns and perform various operations based on those patterns is described.

If you are already familiar with the HP-UX operating system and know which editor you want to use, proceed to the section of interest. If your knowledge of HP-UX is limited, you may want to invest some time in a beginning tutorial such as the book provided with you system, *Introducing UNIX*[1] *System V* by Morgan and McGilton and review the rest of this section.

If you are a newcomer, don't feel intimidated. We all went through the pain of learning at one time. This book will help make the task easier (much easier, we hope) for you than it was for some of us who walked the same path before.

[1] UNIX is a trademark of AT&T Bell Laboratories, Inc.

Which Editor Do I Use?

The HP-UX operating system contains three main editing programs: *vi*, *ed*, and *sed*. However, the *vi* editor program has several independent personalities, each of which is accessed by a corresponding command. The three visually interactive versions of *vi* that maintain a continually updated screen display are accessed through the HP-UX commands *vi*, *view*, and *vedit*. Their differences are discussed briefly below. The three interactive versions of *vi* that do not maintain a continually updated screen display as changes occur are accessed by the HP-UX commands *e*, *ex*, and *edit* (**e** and **ex** are synonymous). *edit* is a variant of *ex* that is sometimes preferred for beginning or casual users, although *vi* (or *view* when file protection is needed) is by far the most popular HP-UX editor among beginners and experienced users alike.

Interactive Editing

Most HP-UX users access HP-UX through a full-feature CRT display terminal or terminal emulator program running on another computer with a full-screen CRT display. For this group of users, *vi* is obviously the most convenient and the most popular HP-UX editor. For the small minority of users who may be restricted to a teleprinter or terminal at the end of a slow (such as a 300 baud) data communications link, a non-interactive editor such as *ex* or *ed* might be more suitable. *vi* has the following variants:

The Vi Editor Family

vi This version is a fully interactive robust editor that maintains a continually updated display screen showing text changes as they occur. The original file being edited is copied into a separate buffer file for editing so that you can make changes to a file without risk of damaging the original until the editing session is finished. With practice, *vi* quickly becomes a versatile, powerful, and convenient editor for performing any operation from the simple to the quite complex.

vi is also able to access *ex* commands for configuring the editor's personality, performing global search-and-replace operations, splitting a file into multiple files, extracting parts of a file, merging text from other files, and other useful tasks. It can also spawn a new shell, enabling you to temporarily suspend *vi*, perform other work, then return without exiting and restarting the editor or losing your place in the file you were editing.

view This version is equivalent to *vi* except that the source file is protected by a **read only** access permission so that it cannot be accidentally destroyed when the edit session is finished. Otherwise, all of the features of *vi* are available. This command is popular among users who need to edit a file in order to create a new file elsewhere, but who want to ensure that the original file cannot be accidentally destroyed in a moment of inattention.

The command **view** is equivalent to the command **vi −R**.

vedit This version of *vi* receives limited use, mainly as a learning tool for beginners who have little editing experience. It is equivalent to using *vi* with the **novice** and **showmode** flags set, and the **report** flag set to **1** (report any changes that affect more than one line before making them). Otherwise, it has all the normal features of the standard **vi** command

e This command is synonymous and interchangeable with the **ex** command if the link has been set up in directory */usr/bin* by your system administrator. The **e** command exists on some early versions of HP-UX, but the single letter violates the rule that all UNIX commands must have two or more letters in their names.

ex This command accesses the non-visual version of the *vi* editor program package. It is interactive with the user, but does **not** maintain a continuously updated display. It is most commonly used on slow terminal devices (such as teleprinters that are considered largely obsolete in the computer industry). *ex* commands are easily accessed from *vi* when using a terminal, so for most users, there is little advantage in using *ex* instead of *vi*.

edit This variant of *ex* is sometimes preferred by beginning or casual users instead of *ex*. However, for those who have a full-feature terminal, *edit* is not commonly used.

The Ed Editor

ed could easily be called the "original recipe" UNIX editor that has descended from the early development stages of the UNIX operating system. Its use is very limited except in shell scripts and other programs, so it is of little interest to most casual users of HP-UX.

ex is an expanded version of *ed* that was later expanded further to form the full-feature *vi* editor. However, a separate tutorial in this volume documents the *ed* editor for those who need to use it.

Non-Interactive Editing

HP-UX also provides a non-interactive streaming editor that is accessed by the **sed** command. This editor uses a command string provided as part of the HP-UX command that starts the editor or a file containing one or more editor commands, then processes the specified text file(s), one line at a time, executing each command on every line as it is encountered in the file(s) being edited. **sed** is capable of processing many large files very quickly when compared with typical hand editing with a typical interactive editor such as *vi*. It is best suited for repetitive operations based on text content or position in a line, and is particularly useful when reformatting text or changing format coding such as converting *nroff/troff* text source into SGML or some other mark-up language for electronic publishing or comparable functions in other occupations.

Table of Contents

Regular Expressions

Introduction

Regular expressions are a simple pattern-matching language used by most major HP-UX text processing tools for locating desired text patterns in a file. They can be used to locate a misspelled word, find all 5-letter words in a file that begin with T or t, locate lines in a file that contain a certain pattern of characters (or a given word) followed by an arbitrary string of text which is followed, in turn, by another specific pattern of text characters, or almost any other imaginable combination. Indeed, the usefulness of regular expressions is frequently limited only by your imaginative abilities as will become evident through further study of the topics in this tutorial.

This tutorial explains how to define and use regular expressions to conduct pattern searches when using an editor or text processor. The subject matter presented here is somewhat tedious and usually of limited interest to beginning users. However, as you gain experience in using the various programs and commands presented in other parts of this manual, you will frequently encounter the need to understand regular expressions at varying levels of complexity. We suggest that you peruse this topic only to the extent of your current interest but be familiar with its content so that you know what capabilities are available when you need them.

HP-UX editors (such as *vi*, *ex*, *ed* and *sed*), text processors (such as *awk* and *grep*), and other HP-UX facilities use user-supplied character sequences called **regular expressions** (or **RE**s for short) to search text files for any character patterns that match the possible character sequences defined by the regular expression. Like the HP-UX operating system itself, regular expressions tend to intimidate newcomers and inexperienced users who have not discovered their incredible ability to quickly locate both simple and tedious character sequences so that needed information or changes can be made with minimum effort. Regular expressions are especially helpful when handling particularly complex operations commonly encountered in sophisticated text edits such as search-and-replace or global changes, or when using *ed* or *sed*. However, if you are willing to invest the necessary time to learn how to use REs, you will find the effort well rewarded. For example, a few hours spent learning how to effectively use REs in *sed* command scripts can save literally weeks of tedious interactive editing with *vi* when the task is large and complex but of such a nature that *sed* can handle most of the job in a few minutes.

Why Use Regular Expressions?

One of the most common tasks for a text editor or text processing program is to scan a file or block of text for a certain sequence of characters prior to performing an insert, append, delete, copy, or replace operation. The character sequences can vary from very simple to extremely complex.

For example, suppose a file contains the word **cjt** which happens to be a typographical error that should have been **cat**. Searching the file for a character sequence that matches the regular expression **cjt** quickly locates the misspelled word, and a simple substitution of **ca** for **cj** solves the problem quite easily.

On the other hand, consider this problem: A source file contains *nroff/troff* font-change commands **\fI** (change to italic font) and **\fR** (change back to Roman font) throughout the text. You want to change the file to a format that uses the standard *mm* (memorandum macros) macros **.I** and **.IR**. This situation poses several problems. First, you have no control over what characters lie between the **\fI** and **\fR**. You know that there is usually only one word between each pair of font changes and that the **\fI** and **\fR** nearly always both appear in pairs on the same line. In addition, the **\fR** may or may not be followed by a punctuation character such as a period, comma or semicolon. A sequence like this:

```
. . .running text \fIword\fR more running text. . .
```

must be changed to

```
. . .running text
.I word
more running text. . .
```

and this sequence:

```
. . .running text \fIword\fR.  More running text. . .
```

must be changed to

```
. . .running text
.IR word .
More running text. . .
```

Several characteristics soon become apparent in this problem:

- The **\fI** is nearly always preceded by a space or appears at the beginning of a line.
- The **\fR** occurs in several positions:
  - At the end of a word before a space character,
  - Between the word and a period, comma, colon, or semicolon,
  - Between the word and end-of-line, or
  - Between the word and one or more spaces or tabs at the end of the line.

We will not solve the problem here, but you can easily see that identifying which sequence is being dealt with and determining what action to take is not as trivial as fixing a three-letter misspelled word.

Here is another problem: Suppose you want to use an editor such as *vi* or *ex* (or *sed* if you are performing the operation on several files in a single batch) to remove all white space at the end of every line in a file (or files) if it exists. If you tell the editor to remove all spaces and tabs, the file would probably become unusable. You need some way to tell the editor to only look for spaces and/or tabs at the end of the line after the last visible character.

Regular expressions provide an easy-to-use and understand (once it is clearly explained) method for describing any type of character sequence or pattern so that it can be correctly identified by the text editor or processor before taking a specified action. Of course, when multiple successive operations must be performed in a given area of text, care must be exercised in choosing the sequence of operations. The pattern recognition and text alteration performed by one operation must not sabotage the success of a subsequent operation by altering or destroying a pattern that the next operation must be able to recognize in order to perform its work correctly.

Where are REs Used?

Regular expressions are used by several popular HP-UX commands/programs:

- *vi* and its related editors *edit*, *ex*, *vedit*, and *view*.
- The *ed* editor and its restricted-access counterpart *red*; also *bfs*, a close relative of *ed*, that is used for searching (scanning) unusually large files for an expression.
- *sed*, the streaming editor.
- *grep* and its relatives *egrep* and *fgrep*.
- *more*.
- *awk*.
- *expr*.
- *lex*.
- *nl*.
- The **−n** option of the *acctcom* command, and
- *bs* in certain situations.

Regular expressions are also used in connection with the system subroutines *regcmp* and *regex*, and in the compile and match routines discussed on the *regexp*(3) manual page in the *HP-UX Reference*.

When Are Regular Expressions Not?

This heading is intentionally confusing. The software that makes up the HP-UX operating system and other similar systems has come from many sources with many authors. How each program handles regular expressions varies somewhat, depending on the program and the operating environment. As some programs from one vendor (University of California, Berkeley, for example) have been merged into the AT&T System V UNIX package, as in the case of *vi*/*ex*, a few minor changes have been made in some instances for various reasons. This factor combined with the natural evolution of programs between releases has led to differences in minor details even though the programs appear largely unchanged to the non-expert user.

As a direct consequence of this confusion amid standardization, the various ways in which regular expressions are handled by various programs has led to attempts by various stardization bodies such as the IEEE POSIX committees and the X/OPEN internationalization committee to resolve the matter more fully. At the time of this writing, there are three general categories of interpretation of regular expressions, with most of the differences largely of little interest to the non-expert user. As the standards issue reaches a fuller resolution, it will be easier to include a complete set of information in this manual.

Defining the Search Area

Some HP-UX commands such as *grep* search entire files for a regular expression while others, such as most editors, can limit searches to as little as one line or search the entire file, depending on the specified address range and nature of the command that includes the regular expression. In this tutorial, searches are usually referred to as extending throughout the file because that is how they are most commonly used. However, the same principles apply whether the search is throughout a long file, many lines in a file, a few lines, or even only one line.

Simple Matches

In their simplest form, regular expressions define a character sequence, character-by-character, exactly as it is expected to look in the text being searched for a matching string. Thus, the regular expression:

```
Now is the time
```

tells the search program to scan its input text for the text pattern `Now is the time` on a single line with or without additional text. On the other hand, if a period and asterisk character pair is introduced between `Now` and `time` in this manner:

```
Now.*time
```

where the period represents any arbitrary character except newline (end-of-line) and asterisk represents zero or more of the preceding character (this combination is often pronounced "dot-star") the expression then can successfully match any of the following text patterns provided each pattern appears on a single line, with or without other text:

```
Nowtime
Now's the time
Now is not the best time
Now is not any time
Now is the worst time
```

and so forth. In this example, the period tells the text scanner to look for any character except newline (end-of-line) in the position represented by its placement in the regular expression. The asterisk tells the scanner to accept any number in succession (zero or more) of the immediately preceding character (or single-character expression as described later). Thus the two together match zero or more successive arbitrary characters except end-of line. The use of the **.*** combination is shown in several later examples along with other uses of the period in the same editor command.

Shell Meta-Characters

It is important at this point that you clearly understand that the use of characters used in regular expressions to represent other characters in text is not related to the shell interpreting special characters as representations of something else. For example, in a regular expression, **$** represents end-of-line, whereas the shell interprets it as a command to substitute the contents of a variable in place of the present parameter in the command (parameter substitution), as in $HOME which tells the shell to use the contents of the shell variable named HOME. Likewise, the asterisk in a regular expression indicates zero or more of the previous character or single-character expression in succession whereas the shell interprets the asterisk to mean zero or more arbitrary characters, such as when identifying a file name.

These two examples show extreme difference and some similarity. Some characters have nearly the same meaning when interpreted by the shell as when they are processed in a regular expression. However, if you clearly establish the understanding in your mind that shell special characters are interpreted in a completely different environment (much like a foreign language in a foreign country) and have nothing to do with regular expressions (despite some similarities), learning how to use them is much easier.

Regular Expressions versus Editor Commands

Another area of common confusion lies in the use of certain characters in regular expressions and using the same characters in other parts of an editor command. For example, the period (.) character represents any arbitrary character except newline when it appears in a regular expression, but represents the current line when it appears in a line address. This becomes difficult for some learners when they see an editor command such as the following *ex* editor command:

```
.,.+3s/^.*[0-9]//
```

where:

- The first dot represents the current line (first address).
- **.+3** represents the third line following the current line (second address).
- **s** is the abbreviated form of the *ex* **substitute** command.
- The first / character separates the regular expression from the command. The second separates the regular expression from the replacement text. The third terminates the replacement text and also separates any command options and/or flags that may follow the replacement text string.
- The complete regular expression used for the pattern search is **^.*[0-9]**, and it is interpreted as follows:
  - The ^ symbol specifies that any matching text pattern must start at the beginning of the line.
  - **.*** represents an arbitrary number (zero or more) of arbitrary characters lying between beginning-of-line (represented by the circumflex) and the last occurrence in the line of any numbers lying in the range zero through nine (defined by the character sequence **[0-9]**).
  - **[0-9]** is a single-character regular expression that specifies a match for any single character in the numeric range of 0 through 9.

Using spoken language, a literal interpretation of this command says, in essence, "Starting with the current line and continuing through the third line after the current line, search each line starting at the beginning of the line and, if you find an arbitrary number of arbitrary characters followed by a numeric character before the end of the line, replace those characters (including the last numeric character on the line) with nothing." In other words, delete the characters identified by the regular expression.

A more straight-forward way of saying the same thing is, "If the line contains one or more numeric characters, delete all characters from beginning of line through the last numeric character on the line, but don't disturb any subsequent non-numeric characters through end-of-line."

Here is another example, where a second character (**$** this time) has a dual meaning and the period (**.**) is used three ways:

```
.,$s/^.*[0-9].*$/This line contained a number in the range 0-9./
```

where:

- As before, the first dot represents the current line (first address).
- **$** represents the last line in the file (second address).
- **s** is the abbreviated form of the *ex* **substitute** command.
- The regular expression is the same as before, except that a sequence **.*$** has been added at the end.
  - The **.*** sequence means, as before, an arbitrary number of arbitrary characters.
  - The **$** represents end-of-line, meaning an arbitrary number of characters up to end-of-line.
- Any text pattern that matches the regular expression is replaced with the replacement text between the last two slash characters in the line.

Note the use of the period as an address, then as an arbitrary character twice in the regular expression, then as a replacement character at the end of the replacement sentence. In the regular expression, the period represents an arbitrary character. If you were searching for a text pattern containing a period in a specific position such as a period at the start of a text formatting macro at the start of a line, the period must be **escaped** by a backslash character to protect the period from being interpreted by the regular expression compile and match operations associated with the editor. For example, to form a regular expression to match a *tbl* table-start macro **.TS** where the period is always in the first column in the line, construct a regular expression as follows:

```
^\.TS
```

Failure to use the backslash causes the interpreter to also match such occurrences as **ATS**, **oTS**, **?TS**, etc., etc. since the period can represent any character except end-of-line (also called newline).

The previous example is equivalent to saying, "Replace any line containing a digit in the range of 0 through 9 with the replacement text."

Constructing Regular Expressions

As indicated in the earlier topic on simple matches, the most elementary form of regular expression is a series of common typing characters that restrict matching to an identical character in an identical sequence in the file being searched. Thus, **cjt** in a regular expression matches **cjt** occurring anywhere in the specified region in the file. However, there are other times when text can be classified into general patterns that do not have identical contents. For example, consider the following series of lines that result from execution of an HP-UX *ll* command:

```
drwxrwxrwx   2 hank     projA       1024 Dec 29  1987 proj_mail
drwxr-x---   2 hank     projA       1024 Oct 25  1987 proj_status
-rwxr-----   2 hank     projA       1024 Jan 24  1988 do_today
drwxr-x---   2 hank     projA       1024 Oct 20  1987 master_files
drwxrwxrwx   2 hank     projA       1024 Dec 20  1987 prod_input
drwx------   2 hank     projA         64 Nov 15  1987 personal
```

Each line is arranged in a columnar format. Some columns have identical text on every line while others vary greatly. Suppose you needed to modify lines that describe directories. It is easy to identify lines that are directories because a **d** is present in the first column. However, if you needed to list directories that included write permission for users outside of the group named **projA**, it becomes somewhat more complex. Here is how each situation can be handled in an expression:

To identify a character at the beginning of a line, it must be preceded by a circumflex character (^) like this:

^d

This expression tells the search algorithm to look for the letter **d** immediately following an imaginary zero-width character at the beginning of each line (represented by the circumflex character) in the text being searched. Since this expression looks at only the first visible character on the line, no other information is available for additional scrutiny.

To find the directories that have write permission enabled for users ouside of the group named **projA**, the letter **w** must be present in column 9. Since we do not know or care what other permissions are set for the directory, we can look at column 1 to find the directory (must match **d** as before), and column 9 for a **w**. We can use the period character to represent arbitrary text for other characters in columns 2 through 8, thus forming the expression:

^d.....w

Finding Long Lines

You can find the lines in a file that contain, for example, more than 50 characters by searching for a pattern matching a regular expression composed of 51 dots. If 51 (or more) arbitrary characters are present in the line, a match occurs. However, if you are using *sed* which can have multiple lines in the pattern space at the same time, a match occurs if 51 or more characters occur between beginning of pattern space and first embedded newline, between last embedded newline and end of pattern space, or between two embedded newlines.

There are many more possibilities, but first, let us introduce the other characters used in constructing regular expressions.

Single-Character Expressions

All regular expressions are constructed from a series of one or more single-character expressions. Single-character expressions can take several forms as indicated in the following list:

Classes of Single-Character Expressions

Character Class	Characters in Class	Description and Use
Typing Characters	**A-Z**, **a-z**, **0-9**, **!**, **@**, **#**, **%**, <, >, (,), {, }, [,], ,, ~, |, :, ;, ?[1] +, =, −, _, **tab**, **space**, control characters	Any alpha-numeric or symbol character that can be typed on a standard terminal keyboard except characters used in substitution representations. These characters match only an identical character in the text being scanned.
Substitution or Search Control Characters	., ^, **$**, /, [,][2], \, *, and −[2].	These characters represent another character, beginning or end of line, or serve as delimiters, range identifier, or escape character in the construction of regular expressions. However, under certain conditions, − and] are interpreted directly as explained in footnote 2 below.
Sets or Ranges of Characters	[⟨set_of_characters⟩] or [⟨range_of_characters⟩] or [⟨combination_of_both⟩]	A group of single characters or range of characters enclosed within a pair of square brackets (such as [**actz58&**] or [**3-7**]) is a single-character expression where a match is accepted if any of the characters between the left and right bracket or in the specified range appears in the position defined by the position of the single-character expression in a larger expression. The second form example shown accepts a match if any one of the numbers 3, 4, 5, 6, or 7 appears in the position indicated.

[1] ? must be preceded by a backslash (\) if used as the first character in a backwards search in *vi* where the search command character is also a ?.

[2] The hyphen is interpreted as a range specifier when defining sets of characters as in the single-character expression [**a-z**] unless it is the first character in a set of characters as in the expression [**−abdfh12**] which matches any one of the characters −, a, b, d, f, h, 1, or 2. Likewise, the right square bracket (]) terminates the expression unless it is the first character in the set as in the group [ ]=+**rt12**] which matches any one of the characters], =, +, r, t, 1, or 2.

Using Typing Characters as Single-Character Expressions

The simplest from of regular expression is a series of one or more typing characters in succession as in a word of text. For example, the regular expression **copiler**, which consists of seven single-character expressions, can be combined with an editor search command to locate a misspelling of the word **compiler** in a file. In general, this form of regular expression is constructed by simply typing a pattern, usually preceded and followed by a slash character (/) to delimit it from other parts of the editor or text processor command.

While quite useful and frequently employed, this form of regular expression has many limitations, especially where multiple patterns having related characteristics need to be located as described in the topics which follow.

Using Beginning/End-of-Line Anchors in Regular Expressions

It is one thing to locate the word **The** anywhere in a text file. It is quite another thing to locate the word **The** when and only when it is the first (or the last) word on a line; especially if, in addition, you don't want the search to match the word **There** or **Then** at the beginning of a line.

Two characters are reserved for use in regular expressions to represent the beginning and the end of a line. The circumflex (^), sometimes called "hat" or (incorrectly) "caret", represents a zero-width imaginary character at beginning-of-line, provided it is the first character in the expression. The dollar sign (**$**) likewise represents a zero-width imaginary character at end-of-line, provided it is the last character in the expression. When constructing a regular expression, the ^ or **$** is typed as a single-character expression just as any normal character except it must be the first or last character, respectively, in the expression. In a nutshell,

^⟨expression⟩	searches for ⟨expression⟩ at beginning-of-line, whereas
⟨expression⟩**$**	searches for ⟨expression⟩ at end-of-line.

Both ^ and **$** must appear as first or last character, respectively, in an expression in order to be interpreted as beginning- or end-of-line substitution characters. If they appear elsewhere in the regular expression, they are interpreted literally as ordinary typing characters. Thus the expression **$The** causes a search routine to search for a four-character sequence consisting of the four visible characters **$The** followed by any arbitrary combination of characters and/or end-of-line.

For example, to find the three letters **The** at the beginning of a line, the correct regular expression would be **^The**. In this example, the circumflex tells the search algorithm to look at the beginning of each line for the character **T** followed immediately by the characters **h** and **e** (if no match occurs, the search routine immediately skips to the next line). To prevent matching other words such as **Then** or **There**, add a space after the **e** in **The** (assuming that **The** is always followed by a space character when in appears at the beginning of a line in the file. This forces the search routine to look for a space after the word before it accepts the match.

Likewise, to locate the same word at the end of a line, use the expression **The$**. This tells the search routine to look for **The** followed immediately by end-of-line. In this instance, unless there is an unusual condition in your text file, it should be unnecessary to begin the expression with a space character since it is unlikely that the word would not be preceded by a space since the initial **T** is uppercase.

Suppose you used an expression such as **$The** or **end$$**. What then? In the first case, the search routine looks for the characters **$The** anywhere on any line being searched. In the second case, the search is for **end$** at the end of any line in the search area. Similarly, **The^** causes a search for **The^** anywhere on any line in the file, while **^^The** searches for **^The** at the beginning of each line.

Representing Arbitrary Characters

If you are a frequent user of text processors, you will commonly encounter the need to search for a group of text objects that closely resemble each other, but which also have important differences. A simple example might be a search for **these** and **those** where only one letter is different. A search for **the** picks up many unwanted words, but by using the dot (.) character (period) to represent any arbitrary character except newline (end-of-line), the regular expression **th.se** matches both words. However, it also matches **th se** in the word-pair "both secondary [schools. . .].

This is easily overcome by using word delimiters if they are available (*vi* only). To find the word **the** when using *vi*, simply surround the word with the character pairs \< and \> as in **\<the\>**. The matches the conditions **^the** (the at beginning of line and followed by a blank, **the$** (the at end of line preceded by a blank, and **the** preceded and followed by blank elsewhere in the line where blank is a space or a tab character. Unfortunately, only the *vi* editor is able to identify words by this method. Other editors require tests for each of the three conditions.

IMPORTANT

Three terms are used when discussing whitespace. **Space** refers to the ASCII space character; **tab** refers to the ASCII horizontal tab character. The term **blank** refers to either a tab or a space.

Using Substitution Characters in Expressions

The most commonly used substitution character is the period and a combination of the period and asterisk as described earlier. Other combinations of substitution and typing characters make up the great majority of regular expressions in most operations. Other examples are included in the *vi/ex* and *sed* tutorials as well as in the *ed* tutorial and elsewhere.

Using Control Characters in Expressions

Characters in this group add dramatic flexibility to your collection of available text processing capabilities. Two very useful character groups are the right and left brackets used to define ranges of characters and character sets to match a given character position, and the left and right parentheses, each preceded by a backslash to construct subexpressions as detailed in the *sed* tutorial and also shown to a lesser extent in the *ed* tutorial.

Let's talk about the right and left brackets. Suppose you need to isolate a pattern such as the filename that contains the *HP-UX Reference* entry for the *fsck*[*SDF*](1M) command in a long list of filenames as well as any other manual page that is for the SDF file format. Assuming the names all start at the beginning of the line and end with whitespace, the regular expression looks like this:

```
.*\[SDF\]
```

This form also matches sections other than 1M if applicable. Note the use of backslash before square brackets to force interpretation as typing characters instead of control characters.

Now let's change the rules so that any filename that has three uppercase letters between square brackets is matched. Here is the expression:

```
.*\[[A-Z][A-Z][A-Z]\]
```

This expression matches a square bracket followed by a letter in the range A-Z followed by another followed, in turn, by another. A closing square bracket finishes the expression matching.

Suppose you want to match the words: that, they, thou, thee, but not them. You can specify certain characters that can be accepted as a match in each position by placing the characters between square brackets as follows:

```
th[aeo][tyue]
```

However, this expression also matches the words thay, thau, thae, thet, theu, thot, etc. These examples show the need for judgement in setting up regular expression groups and ranges to obtain correct results.

Excluding Characters From a Set

You can also specify that any character is to be accepted as a match **except** those specified by starting the series with a circumflex and placing the series between square brackets. For example,

```
[^aslm]
```

in a given position tells the matching routines to accept any character in the position represented by this single-character expression except **a**, **s**, **l**, and **m**,

Other combinations are possible. Refer to the examples in the *sed*, *ed*, and *vi*/*ex* tutorials, and to the *ed*(1) manual page entry in the *HP-UX Reference* for additional information.

Table of Contents

Ex Command Descriptions (Chapter 10 continued):

Chapter 12: Editor: Configuring the Vi/Ex Editor

All You Need to Know

It has been said that *vi* is hard to learn. Not so. Everything you need to know is on this and the next page.

Vi the Lazy Way

If you quickly thumb through this tutorial, you will discover that several hundred pages were needed to detail *vi*'s many capabilities and show how to properly use them. If you want to become a proficient *vi* user, you will find a thorough study of its contents very rewarding. However, on the other hand, if you think you are too busy to take the trouble, you can get by using only 12 basic commands. They are summarized here for those who insist on doing it the "quick and dirty" way. You'll be much less efficient, but you can get simple jobs done. Then, when the pressing matters of the moment are less urgent, you can study the manual in detail and discover how much time you could have saved by using better methods. Here they are:

Command Mode Access Commands

ESC	Return to Command Mode. Used to terminate adding new text after **i**, **a**, **o**, and **O** commands. If you press ESC and get a beep, you are already in command mode.
:	Start *ex*-mode command at bottom of screen. A few ommands are listed on the next page.

Vi Editor Commands

i	Insert new text in front of current character. ESC ends insertion.
a	Append new text after current character. ESC ends appending.
x	Delete current cursor character.
dd	Delete entire current cursor line.
h	Move cursor left one position.
l	Move cursor right one position.
j	Move cursor down one line.
k	Move cursor up one line.

More Vi Editor Commands

o	Open a new line after current line and start inserting new text until ESC is pressed.
O	Open a new line before current line and start inserting new text until ESC is pressed.
ZZ	Terminate the editing session after storing the buffer file in permanent storage.
/text	Search file for series of characters that matches *text.*

Ex-Mode Commands

:w	Write file to *filename.*
:q	Quit if the file has been preserved in permanent storage.
:q!	Quit after discarding edited file. Do not store it permanently.

Easier Ways

r	Replace current cursor character with a new single character.
R	Replace existing text, one character at a time, until ESC is pressed.
u	Cancel the last change and restore to condition before change.
U	Restore current line to original condition before you changed it.
Y	Yank (copy) current line into the unnamed buffer.
p	Put unnamed buffer contents into text after the current cursor position.
P	Put unnamed buffer contents into text before the current cursor position.

Introducing the Vi/Ex Editor

1

The HP-UX operating system contains a powerful text editor program that exhibits several personalities, depending on which command was used to start it. However, of the six possible commands, only two really distinctive personalities exist:

- The *vi*, *view*, and *vedit* commands access the interactive style of operation that maintains a continuously updated screen display that shows changes as they occur.
- The *ex* and *edit* commands access the line-editor style of operation which is essentially an extended version of the UNIX[1] line editor program *ed*.

HP-UX Command Names for the Vi Editor

vi Full-capability visually interactive editor that maintains a continuously updated display screen showing the changes made by the editor as they occur.

view Equivalent to *vi* except that the original file is marked as "read-only" to the editor so that it is reasonably safe from being casually or accidentally overwritten or destroyed[2] by the user or the editor during the session. However, the edited file can be stored elsewhere in a new file. This command is commonly used when original source files must be carefully preserved.

vedit Same capabilities as *vi*, but special flags are set, indicating to the editor program that the user is a beginner who wants special treatment. Various releases of *vi* have differing default options for *vedit*, making the behavior of this command somewhat unpredictable. Most users, including casual and inexperienced, use the standard *vi* and *view* commands. Few HP-UX users ever use *vedit*.

[1] UNIX is a trademark of AT&T Bell Laboratories, Inc.

[2] The original file can be overwritten by a forced write command if you have the necessary access permissions on the file.

HP-UX Command Names for the Ex Editor

ex Extended form of the *ed* editor. Not as popular as the *vi* editor because it does not maintain an updated display of edited text as the session progresses and most *ex* features are readily accessible from *vi*.

edit Counterpart of *vedit* for the *ex* editor. Editor behavior is altered to accommodate the needs of casual or inexperienced users (sets **novice**, **nomagic**, **report=1**, and **showmode** which are described in Chapter 12). This form of *ex* holds little or no interest for most HP-UX users.

vi and *view* are by far the most commonly used editors among HP-UX users equipped with CRT display terminals. *ex* is used much less frequently. *vedit* and *edit* are rarely used on most systems, mainly because *vi* is reasonably straightforward and easy to learn and use, despite its powerful capabilities.

Switching Between Vi and Ex

There are times when it is necessary or desirable to switch from *vi* to *ex* editor personalities and vice-versa without leaving the session. This is easily accomplished by pressing **Q** (SHIFT-Q) when in *vi* to change to *ex* or typing **vi** RETURN after the colon prompt when in *ex*. This situation most commonly arises when performing intermediate and advanced editing tasks as described in the more advanced chapters of this manual.

For Ex Users

If you are using *ex* exclusively instead of *vi* (perhaps because you have an electromechanical printing terminal instead of a CRT display terminal), most of the mechanics of using *ex* are explained in Chapter 13, and in Chapters 10 and 12 with some topics of interest scattered elsewhere such as in Chapters 6 and 7. However, this manual is designed to be most useful for those who have access to CRT display terminals (well over 95% of all HP-UX users).

International Language Support

vi and *ex* support 8-bit character codes necessary for editing text files in various languages. HP-UX supports several 8-bit character sets that are described in Section 5 of the *HP-UX Reference* such as *kana*(8) and *roman*(8). This means that the *vi*/*ex* editor does not strip the eighth bit from ASCII text, unlike some similar editors having the same name on various UNIX[1] or UNIX-like systems. For 8-bit character sets to be correctly displayed on a terminal, system terminal support must be correctly configured for language and character set being used.

vi on HP 9000 Series 800 and Series 300 systems also supports 16-bit[1] (Asian) languages, provided the proper local language option is installed and operational.

For more information about using *vi* in non-English 8- and 16-bit language environments, refer to the documents included with the optional local-language software.

[1] Portions of the 16-bit capabilities in *vi* are based, in part, on software developed by the Toshiba Corporation.

Why Vi?

The *vi* command provides a powerful, visually interactive text editor that provides a continuously updated text display as editing progresses. The close interaction between user and editor program includes the ability to recover from mistakes by using the editor's "undo" command. Since *vi* is an extension of the *ex* editor, it also supports many *ex* capabilities that simplify repetitive operations such as search-and-replace and global changes on all or part of a file from a simple command, provide typing aids with the abbreviate command, and support file manipulation and other capabilities. These conveniences, coupled with several safeguards to aid in recovery from operator errors, provide a flexible tool that meets the needs of beginning and experienced users alike.

The *ex* command accesses a useful, but much less commonly used, editor that does not provide the high level of visual interaction with the user that is available from *vi*. *Ex* is an extended form of the early UNIX editor *ed* that receives only limited use on most systems (other than in shell scripts and programs). While both *ex* and *vi* are really only a single program that behaves differently depending on the command used to access it, where, when, and how each command is used varies significantly.

Audience Definition and Learning Suggestions

This manual is designed so that anyone who has a rudimentary understanding of the HP-UX file system and text editors in general, but little experience with HP-UX, can quickly begin performing useful work using *vi*, then progress with study and experience to an expert level. Because of the broad range of skill and knowledge that is addressed, the tutorial is, of necessity, rather lengthy. However, few users will have a need to read the entire manual from start to finish.

Use the Table of Contents to grasp the general outline and structure of the manual, then select those areas you need to understand and proceed accordingly. Beginning users should read the first four chapters and try the examples on a terminal before proceeding further.

If you are a more experienced user, you will likely discover that time spent perusing topics you are already familiar with will yield enough useful techniques that you had previously overlooked to make the effort worthwhile. Considerable effort has been invested in providing various unusual shortcuts to easier editing by bringing some of *vi*'s largely undiscovered capabilities out of obscurity.

Manual Organization

The bulk of this tutorial explains how to use the *vi* editor and access and use the *ex* commands from *vi*. A separate chapter explains how to access and use the *ex* editor, then refers to other chapters for detailed information about *ex* editing commands. The chapters that form the body of this text are tutorial in nature. Appendices at the end contain abbreviated reference material of value to experienced users. The Table of Contents provides a useful list of the topics covered in this manual, and the index is more comprehensive than is typical in most UNIX-like systems documentation.

User Interaction

vi has three main operating modes and a few related behavior traits that may be somewhat confusing to a beginning user. They are presented here for reference. You may prefer to skim over this section so that you know what it contains, then come back later when understanding it becomes important.

Operating Modes

The three primary operating modes are

- Text-input mode,
- *vi*-command mode, and
- *ex*-command mode, sometimes referred to as external mode.

Associated with the third (external) operating mode is the shell-escape command that is used to access HP-UX system commands as well as the external capabilities in the *ex* editor that are used by *vi* for such tasks as global changes, search-and-replace, and other operations on all or part of a file.

Not to be confused with the external-mode shell escape is the *vi* command mode shell escape that pipes all or part of the buffer to an HP-UX command. This feature is used for performing paragraph adjusting, sorting, and other useful tasks on part or on all of the current *vi* buffer. This topic is discussed in the chapter entitled Advanced Editing: Shell Operations

Vi Command Mode

When *vi* is started and ready for use, Command Mode is active, and all keyboard input is treated as command information until a valid command is encountered. There are two classes of commands: non-printing, and printing.

- Most *vi* user commands are non-printing, which means that they are not printed on the terminal as they are received by *vi* because they would disrupt the visible text display. Such commands are usually related to inserting, deleting, or altering text relative to the current cursor position, or they pertain to cursor movement or changing text position on the display screen. Processing these commands without printing them on the display screen is not a problem because other visible terminal or display behavior indicates that the command was received correctly. If a mistake is made, the undo command quickly repairs the damage and you can try again.
- *vi* has only four printing commands. They are:
  - /, the forward search command,
  - ?, the backward search command, and
  - !, the shell pipe (or filter) command.
  - :, the prefix for *ex* commands being executed from *vi*.

  These commands and the ensuing command text (search string and/or HP-UX command) are printed on the bottom line of the display screen below the displayed text. These commands are used for such editor tasks as searching forward or backwards for a text pattern, or piping all or parts of the buffer through an HP-UX command (such as *sort*, *adjust*, or *pr*).

vi always uses the bottom line of the screen to display printing commands, error messages, and echoed command lines. Look there to verify your commands or find other information about errors and command completion.

Text Input Mode

vi enters Text Input mode whenever an insert text, append text, or change text command is given. In Text Input mode, characters are added to the text file as they are typed from the terminal keyboard until you press ESC, at which time *vi* returns to Command Mode. Text Input mode is described in detail in later chapters.

If you press ESC while in command mode, the terminal *vi* ignores the command and beeps the terminal.

New *vi* users often find it difficult to determine whether they are in input mode or command mode. The editor can be reconfigured quite easily to display the mode by typing the following command from the keyboard after the editing session is started as described later:

```
:set showmode RETURN
```

Once this command has been typed, *vi* displays the message `INPUT MODE` in the lower right-hand corner of the editor display screen when in input mode. If not in input mode, no message is present. This command can also be placed in your *.exrc* file as explained in Chapter 12 under the topic "Automating Editor Configuration".

External Mode

External Mode is accessed by typing **:** while in Command Mode. The colon, the first character in a printing external command, is displayed on the bottom line of the screen and tells *vi* that the command is to be executed by *ex*. If the colon is followed by an exclamation point, the exclamation point tells *ex* that the remainder of the command is to be passed to an HP-UX shell for execution instead of being processed by *ex*.

Upon completion of any external-mode operation, control is returned to *vi*. When an *ex* command results in a shell escape, the return from the HP-UX shell requires that you press a key telling *vi* to update the display screen and resume operation. This provides an opportunity for you to review any displayed information on the screen (such as the results from a file listing command, for example) before it is destroyed when *vi* overwrites the screen. Whenever *vi* suspends operation for a shell escape, it provides a prompt indicating the proper recovery procedure.

If an open file was modified prior to the External Mode command, *vi* may issue the message:

```
[No write since last change]
```

This message occurs when the command that produces the message results in a shell escape from the editor. It has two purposes: First, it is a caution warning telling you that the current workfile has not been written back to permanent storage and the changes you have made will be lost if no write operation occurs before terminating *vi* upon return from the shell escape; Second, it also tells you that if an HP-UX program such as a compiler is run by the external-mode command, the source file cannot be used because it is not up to date. Since most shell-escape commands result in a return to *vi*, loss of the edited file is not usually a concern unless something happens that causes *vi* to be aborted. However, if you run a compiler such as *cc* on an older version of the file, you will probably not get the desired result. If you need to abort such a command, press BREAK to return to the editor.

When HP-UX completes the shell escape and returns to *vi*, *vi* usually displays the message:

```
[Hit return to continue]
```

Violence directed toward a keyboard is definitely not recommended. Simply press (no need to hit) any normal typing key such as RETURN or the space bar to resume editing. *vi* then updates the screen and places the cursor where it was before *vi* was interrupted by the external command.

What Are All Those Tildes (~) On My Screen?

If you are editing a new file or if text is positioned on the display screen such that the end of the file occurs before the bottom line of the screen, *vi* displays tilde (~) characters down the left-hand column of the display screen. These characters are placed there by *vi* as part of its normal display screen handling processes to mark lines on the display that have no corresponding line of text in the file being displayed. *vi* does not place any visible characters in the text file stored on disk other than those that were intentionally placed there by someone using *vi* at the time the file was created or during a subsequent editing session, so relax – those tildes don't mean a thing, but they can help you recognize blank lines at the end of a text file. If you see empty lines between the last line of visible text in the file and the first line with a tilde in the left column on the display screen, those empty lines are blank lines in the file (or a long string of spaces and/or tabs on the last line that forced the line to wrap to the succeeding line(s) – not likely, but possible).

You may also occasionally see a ~ at the end of the last line in the file. The tilde indicates that the last line in the file is not terminated by a newline character. Normally, this can only happen when the original file being edited had no trailing newline at the start of the session. When the file is written back out at the end of the editing session, *vi* places a newline after the last line.

What about Long Text Lines at the Bottom of My Screen?

Occasionally, you may encounter files where a single line contains more characters than can be displayed on a single line of the terminal's CRT display. When such a line is encountered, *vi* wraps the line at the right screen margin onto the next line without regard for word boundaries. The line is still a single line in the buffer file, but is displayed as two or more lines on the CRT.

Occasionally, as when scrolling text, a displayed line preceding a long line may occupy the next to the last line of the CRT display window (excluding the command line at the bottom of the screen). Since *vi* cannot display the following line on the single available line at the bottom of the screen because it is too long, *vi* omits the line from the display and, instead, places a single character (@) in the left column of the blank display line. If the undisplayed line requires two or more lines on the CRT display, an @ character is placed at the beginning of each blanked display line until sufficient lines are available to display the entire line of text as it exists in the buffer file.

If a long line is scrolled off the top of the screen, it disappears one display line at a time as the lines roll up. If the screen is scrolled down, the display jumps downward by enough display lines to bring the long line fully into view.

Program Limits

vi and *ex* impose several limits which must be considered when you depart from the range of typical applications. However, these limits rarely affect most users. They are grouped here for your convenience should you need to know what they are. The commands and other conditions that these limits affect are described throughout the later chapters in this tutorial.

Maximum Line Length

vi allows line lengths up to 1024 characters including a small number of characters (about two or three) used for overhead. In general, unless your line lengths can exceed 1020 characters, you should have no difficulty. .index: Maximum line length

Maximum File Size

On Series 500 systems, hardware constraints limit maximum file size to about ½ megabyte. Exceeding the limit causes a memory segmentation error.

On Series 300 and 800 systems, system capacity is such that file size is rarely, if ever, of concern since the silently enforced maximum file length is 250 000 lines.

Other Limitations

Other limits you are likely to encounter as you reach more advanced levels include:

- 256 characters per global command list,
- 128 characters in a file name (HP-UX maximum filename length is 14 characters) in *vi* or *ex* **open** mode,
- 128 characters in the previous-insert/delete buffer,
- 100 characters in a shell escape command,
- 63 characters in a string valued option (**:set** command),
- 30 characters in a program tag name,
- 32 or fewer macros defined by **map** command,
- 512 or fewer characters total in combined existing **map** macros.

Basic Editing

Editing consists of various related tasks that include:

- Creating new text; called text entry or text input
- Deleting existing text
- Altering existing text
- Inserting or appending new text into an existing line
- Inserting new lines between existing lines
- Rearranging blocks of existing text

The next few chapters describe relatively simple editor tasks that are commonly used by a majority of users. More advanced topics are discussed in later chapters. A summary of commands, key functions and other information is located at the end of this tutorial for easy reference.

Basic Editing: Starting and Ending a Session 2

An editing session begins when you invoke *vi* from an HP-UX user shell (or program). The session can be terminated normally by a write-and-quit command, or it can be aborted by using only the quit command in which case the results of the editing session are discarded unless they were previously saved by a separate write command. This chapter describes each of these possibilities in greater detail.

File Usage During an Editing Session

When *vi* starts operation, it creates a buffer file (in the system directory */tmp*) that is used to hold the text being edited. If you are editing an existing file, the existing file is copied into the buffer for editing. If you specify a new file to edit, new text from the terminal keyboard is copied into the buffer file and edited according to the editing commands you provide. At the end of the session, unless you specify otherwise, the contents of the buffer file are copied back into the existing file, or a new file is created to hold the buffer contents if you are editing a new file.

Baud Rate versus Display Size

On most HP-UX systems, the terminals are wired directly to the computer or use high-speed modems. Such installations usually display a full screen on the terminal display; typically 23 lines plus a command line at the bottom of the screen. However, when a slower modem is used for connection over public telephone lines, the time required to draw the screen when opening a new session or redrawing after a jump to a new location in the file, can become disconcerting. To minimize delays in updating the display, *vi* displays fewer than 23 lines on baud rates of 1200 or less, then adds more lines as editing progresses in the new file area. To further conserve time and resources, *vi* always determines and uses screen updating methods that require the fewest possible characters to make the needed display changes. Partial screen displays are discussed in greater detail in Chapter 12 under the **window**, **w300**, **w1200**, and **w9600** options. If you are using a slow modem connection but still prefer a full screen, you can override the default values on these options by setting them to non-default values as explained in Chapter 12.

Opening a Session

The editing session begins by invoking *vi* with an optional filename. If no filename is specified when you start the session, you must end the session by using a write or write-and-quit command with a filename provided at that time. Thus, it is usually preferred (and easier) to provide the filename when *vi* is invoked. If you want to edit an existing file, it is much easier to provide a filename at the beginning than to use read commands to retrieve the file after *vi* has started.

Selecting the File

vi can be used to edit an existing file or create a new file. In either case, the name of the existing file or the name of the new file to be created is usually specified when invoking *vi* as follows:

`vi` *filename* RETURN

If you intentionally or inadvertently invoke *vi* without including the filename, *vi* opens a buffer file that is maintained for the duration of the editing session. Upon completion of (or at any time during) the session, you can save the contents of the buffer file in a specified *filename* by using the Write command that is described later in this chapter. If you prefer, you can abort the session at any time and destroy the contents of the buffer file without disturbing the original contents of the source text file being edited by using the Quit command as explained later in this chapter under the section entitled Terminating a Session.

File Must Be a Text File

Only text files consisting of ASCII (or other supported) characters can be edited using *vi*. If you attempt to edit a non-text file, an error will probably result. Non-text files can also wreak havoc with terminal configuration.

Existing or New File?

During startup, *vi* checks to see if *filename* exists. If the file is present in the current (or specified other directory), it is copied to a buffer file, then the beginning part of the buffer file is printed on the display screen. (If the file is smaller than the available screen size, the entire file is displayed and any unused display lines are marked by tildes in the left-most column.) When *vi* opens an existing file for editing, pertinent information concerning the filename and number of lines and characters in the file is displayed at the bottom of the screen. During the edit, only the buffer file is used; the original file is preserved in its original, undisturbed form. Depending on which options you use when terminating the edit session, the buffer file is usually written back to the original file at which time the original file is destroyed and replaced by the new edited file.

If *filename* does not exist (or if *filename* is a directory), *vi* opens a buffer file and displays a blank screen with tilde (~) characters down the left side, one on each line. *vi* then awaits your first command. At the end of the session, again depending on which options you select and what commands are used to terminate the edit, *vi* usually writes the contents of the buffer file to a new file, giving it the *filename* you specified when invoking *vi* at the beginning of the session.

Note the message at the bottom of the screen when the file is opened. If the specified file does not exist, *vi* displays the filename enclosed between double quote marks followed by:

```
[New file]
```

What if filename is a Directory?

If the specified file is a directory, the filename between quotes is followed by:

```
Directory
```

This means that you must take alternative action now or later because *vi* cannot be used to alter the contents of a directory. To allow otherwise would create catastrophic confusion in the HP-UX file system. If this situation arises, refer to the topics under the Common Difficulties section of this chapter for further instructions.

Editing an Existing File

If *filename* already exists, *vi* copies the file into a temporary buffer file, then displays the first few lines of the file on the terminal screen. You can now edit the file as discussed in later chapters. When you have finished editing, terminate the session using any of the methods discussed in the next parts of this chapter as well as elsewhere in this tutorial.

Protecting an Existing File

Most users occasionally need to edit an existing file but need a guarantee that they cannot accidentally overwrite it during or at the end of an editing session. As usual, HP-UX and *vi* provide several ways for doing this. You can use the HP-UX *cp* command to copy the original file to a new file then edit the new file, or you can use the **readonly** option to the *command* which, again, can be handled several ways:

- Start the session by using a **−R** option between the *vi* command and the filename,
- Use the *view* command which is equivalent to using the **−R** option with *vi*, or
- Start the session with the usual *vi* command, then use the **:set readonly** command to the editor to prevent overwriting the file.

These techniques are equivalent. Obviously, the easiest of the three is simply use the *view* command instead of *vi*, then proceed with the edit. When you are ready to save the file in permanent storage, use the **:w** command and a different filename before terminating *vi*. Exact procedures for saving the edited file are explained in detail later in this and subsequent chapters.

Note

Do not fall prey to a false sense of security when using the **:set readonly** or *view* commands to protect your source file. You can still destroy it by using a **:w!** command (no space between **w** and **!**) if you have proper access permissions to the file. This is especially hazardous if you start editing another file while in the current editing session by using a **:vi** *filename* command from *vi* or *view*. The best way to protect a file is by setting the file access permissions to read-only (mode 400, 440, or 444) by using the HP-UX *chmod* command.

Editing Lisp Files

vi has a special option for editing Lisp program files which is used as follows:

`vi -l` *filename* RETURN

When the **–l** option is used, indents and the cursor-move characters (,), {, }, [[, and]] are redefined so that they are compatible with Lisp programming. If the lisp option is active, press % to move back and forth between matching parentheses, etc. If the session is already underway, you can use the **:set lisp** command as described in Chapter 12.

Terminating a Session

Upon completion of an editing session, you have several options:

- Abort the edit and destroy the contents of the buffer file.
- Terminate normally by writing the buffer file contents to the file specified when *vi* was invoked.
- Write all or part of the buffer file to one or more alternate filenames then abort.
- Write all or part of the buffer file to one or more alternate filenames then continue editing.
- Write all or part of the buffer file to one or more alternate filenames then write it to the file specified when *vi* was invoked and terminate normally.
- Any combination of the above as well as other options.

Normal Termination

vi editing sessions are nearly always terminated normally by a write-and-quit command that can be given in either of two forms. With *vi* in Command Mode (if you are not sure, press ESC once or twice until you hear a beep from the terminal), type either of the following:

```
ZZ
```

or

```
:wq RETURN
```

Obviously the first is easier because it is a shorthand form of the second and does not require use of the RETURN key.

The second form is an *ex* command that is executed from *vi*. The **w** tells HP-UX to write the buffer file to the current file that was (usually) specified when *vi* was invoked. The **q** command tells *vi* to abort after the file is written to permanent storage.

Other forms of the write (**w**) command are useful for splitting files, saving parts of a file in other locations, and other related tasks. The write command is described in greater detail later in this chapter as well as in the File Manipulation Techniques chapter of this tutorial.

Note

The **ZZ** command must be used with care when using the **:w** or **:w!** command to copy the working file to other files. If the original file is modified then written to another file, typing **ZZ** terminates the edit without updating the original file. This problem is discussed in greater detail later in this chapter.

Aborting an Editing Session

You may occasionally want to abort an editing session for any of various reasons. Sometimes a complex operation can be disastrously misdirected due to a typographical error or accidental keystroke, making it easier to start over than to repair the damage. At other times, you may simply decide to abort the edit and go back to the original version. You also may use *vi* at times to scan a file then want to abort the session to ensure that the source file is not altered.

You can easily exit gracefully without disturbing existing file structures by using the quit (**:q**) command. If you execute any editing command before quitting, *vi* will not accept **:q** as a valid quit command (this is to keep you from abandoning a long editing session if you accidentally give a wrong command). If you want to quit, even though you may have already executed an editing operation, use **:q!** instead to override the protection barrier.

If you send *vi* an ordinary quit (**:q**) command and an override is needed because one or more editing commands were executed before the quit command was received, *vi* sends the error message

```
No write since last change (:quit! overrides)
```

To force *vi* to quit, repeat the quit command with the exclamation point. If you want to continue editing, just give *vi* your next command. No recovery is necessary. Note that **:q** and **:quit** are equivalent, but most users prefer **:q** because it is easier to type.

Common Difficulties

Typographical errors, oversights, slips of the finger or a brief mental lapse can lead to mistakes, however minor, when invoking *vi*. The consequences are seldom serious, but can be disconcerting if you don't know how to recover. Here are a few examples of typical problems that cover most situations.

Oops! I Got the Wrong File. Now What?

The most common error is usually accidentally striking the wrong key when specifying a filename. This causes *vi* to look for an incorrect file name that may or may not exist. If *vi* cannot find the file, it opens a new file. If a file exists whose name matches the mistyped one, the wrong file is opened for editing.

Another common error occurs when you forget to include a complete directory pathname and the file does not reside in the current working directory. Again, *vi* opens a new file if the name is not present in the current directory, or it opens a wrong file if a file having the same name is present.

When such accidents happen, you usually do not want to modify an existing file, nor do you want to create a new one. To exit gracefully without disturbing existing file structures, use the quit (**:q** or **:q!**) command discussed earlier in this chapter under the topic Aborting an Editing Session.

Time-Saving Tip

When you abort an editing session then restart with a new file, the *vi* program must be reloaded into memory by HP-UX. You can avoid the time required to reload the editor when you switch to a new file by using the command:

`:e` *new_filename* RETURN

This *ex*-mode command causes *vi* to abandon the current buffer file contents and immediately open a new file and reload the buffer without terminating the editor program, usually saving several seconds. If you have modified a file and want to abort and change files without updating the original file and reloading the editor program, use:

`:e!` *new_filename* RETURN

Either of these commands can be used after the **write** command (**:w** or **:w!**) described later in this chpater if you need to edit another file and don't want to reload the editor.

What if I Try to Edit a Directory?

It is easy to absent-mindedly type the name of a directory when invoking *vi*. When *filename* is a directory, you obviously cannot store the buffer file under that filename when you finish. You have two options:

- Abort the edit immediately by typing

  `:q!` RETURN

  or

- Continue the new file then write it to a file when finished by using the write command:

  `:w` *filename* RETURN

  or

  `:w!` *existing_filename* RETURN

then following it with an abort (**:q** or **:q!**) command. The safest and least confusing option is usually to abort immediately and try again with a valid filename.

Note

Do not type a space character between **w** and **!** when using the **:w!** command. If a space is present, *vi* interprets it as a command to copy the buffer to standard input and pipe it to the command *existing_filename* which may not exist, and, if it does, certainly is not what you want (the **:w !** ⟨*command*⟩ command is discussed at the end of Chapter 11).

Starting a New File

If you specify a *filename* that does not exist when invoking *vi*, after *vi* completes its start-up initialization including creation of a temporary buffer file, your terminal will show an empty screen with the cursor in the upper left corner and a row of tilde (~) characters down the left side. The new filename followed by the message `[new file]` appears on the bottom line of the screen, indicating that *vi* is ready for your first command:

```
~
~
~
my_file [new file]
```

Entering Text

To enter text in a new file, press **i** (insert text in front of the character identified by the current cursor position) or **a** (append text following the character identified by the current cursor position). At the time you press **i** or **a**, the editor is in command-input mode, so the command character is not printed when pressed and no visible change in the screen display is evident. However, as soon as you press **i** or **a**, *vi* changes immediately to text input mode so you can start typing in new text. Begin by typing text much as you would with an ordinary typewriter. Other methods for adding new text are discussed in Chapter 4 which discusses text manipulation techniques.

To quit entering text and return to command mode at any time, simply press ESC.

Backing Up over Typographical Errors

When occasional typographical errors occur during text input, an important convenience is being able to backspace, overstrike the error, then continue. *vi* handles the need with only minor inconvenience. Simply use the [BACK SPACE] key or [CTRL]-[H] to back up to the (first) mistyped character, retype it, then retype the text that you backspaced over. The restrictions in this technique are that you can only back up on the current cursor line, and you cannot backspace beyond the point where you started inserting new text. If the error occurred on an earlier line, you must press [ESC] to return to command mode, move the cursor to the error, then make the needed corrections using techniques described in the Basic Editing chapters on display and cursor control and manipulating text.

If the line is long and the error is early in the line, it may be easier to press [ESC], back up to the error(s), then replace with new character(s) than to backspace then retype the rest of the line. Again, methods for changing errors in a line are discussed in detail in the chapter, Basic Editing: Manipulating Text.

Line Lengths

vi can accept line lengths longer than the width of the terminal CRT screen (up to about 1020 characters). If the line length exceeds maximum screen width, *vi* breaks the line at the right-hand screen boundary (often in the middle of a word), and continues it on the next physical line on the display even though the line is treated as a single line in the text file. This feature is helpful for some situations where long lines are necessary, but for ordinary text, shorter lines are much easier to deal with. A good general rule-of-thumb to use for ordinary text is:

- Keep lines shorter than the width of the narrowest terminal that will be using the file (usually about 70 to 75 characters is appropriate for standard 80-character screen widths).
- End each line by pressing [RETURN] (you can also use the **wrapmargin** option that is discussed later in the topic Continuous Text Input to eliminate the need to use [RETURN] after each line).

Limiting line length to a single display line keeps text from confusing other users when they must edit it, especially when moving the cursor up or down through text. For example, suppose a given long line of text in a file occupies three lines on the CRT display while the next line in the file occupies only one line on the CRT. As you move the cursor down the screen, it reaches the first displayed line of the long line in the file. As the cursor moves down to the next line in the file, it suddenly jumps three lines at once on the screen, creating the illusion that it skipped two lines. However, *vi* recognized only the long line **in the file** which happened to require three lines **on the display**. It moved the cursor to the same column of the next line **in the file** which was being displayed on the third lower line on the display.

Continuous Text Input

You can use the *ex* **set** command to create an automatic right margin located a specified number of columns from the right-hand side of the terminal screen. Then, whenever the cursor reaches the column that has been defined as the right-hand margin, the cursor is immediately moved to the left margin of the next line. If a word has been only partly typed when the margin is crossed, the first part of the word is also moved to the next line and the cursor advanced accordingly. To set the margin, type:

```
:set wrapmargin=8
```

This establishes a margin eight characters in from the right-hand edge of the screen (column 72 on an 80-column display). To select a different margin, simply replace the number **8** in the command with some other value that does not exceed the width of the screen in columns.

Once **wrapmargin** has been set, you can begin typing text at will (in Insert Mode, of course) and *vi* will automatically jump to the next line when you exceed the margin. This feature is especially useful to touch typists who are transcribing large amounts of text from a typed, printed, or handwritten source, or users who want to save a few keystrokes. **wrapmargin** can also be automatically set each time *vi* is used by placing a **set wrapmargin** command in a *.exrc* configuration file in your home directory. Using the *.exrc* file is discussed at length in the chapter entitled Configuring the Vi/Ex Editor.

I Have Typed My Text. Now What?

Once you have placed the editor in insert or append mode by pressing **i** or **a**, all incoming characters are treated as new text until an escape character is received from the keyboard. To terminate text entry, press ESC (**ALT** on some terminals), or CTRL-[. *vi* then returns to Command Mode and awaits your next command.

Note Regarding Escape Sequences

Most terminals use escape sequences to implement arrow and function keys (and some other keys, depending on the terminal). These sequences consist of an ASCII ESC character followed by one or more typing characters. Since *vi* uses ESC to terminate text entry, it must be able to differentiate between an ESC to terminate input followed by another key such as **h** for backspacing, and the HOME UP key that also provides an ESC-h character pair and moves the cursor to the upper left corner of the screen. By enabling the **timeout** option (described in Chapter 12), a time limit that varies, depending on which computer series and which HP-UX release you are using[1] is used to determine: that the sequence is a terminate-input, backspace if timeout is exceeded; or HOME UP if the timeout does not expire.

This can be a problem for fast users who type ESC then follow quickly with a new *vi* command such as **h** and see the cursor snapped from its position to the upper left corner of the screen. The operation, in effect, terminates input mode, executes a cursor move to the new location, reopens input mode, and accepts new typing until [ESC] is typed again. Pressing **u** to undo the change on the new cursor line after the ESC h restores changes after the move, but does not affect text before the misinterpreted escape sequence (you can use " (double accent grave) to return to the cursor location before the misinterpreted escape sequence). Consequently, experienced users who use home row keys for cursor control may find ESC followed by a cursor move being interpreted by *vi* as something other than what they intended unless they pause slightly after pressing ESC before using any other keys.

Solving the problem is made even more difficult by the programming demands of international language support, but efforts are being made to resolve or minimize this problem in future releases of HP-UX.

[1] The timeout value was increased to 500 ms from 85 ms at Series 500 Release 5.2 to accommodate certain slow terminals and their users who use arrow and function keys).

Saving Text During an Edit

As described earlier in this chapter, *vi* maintains a buffer file that is used to hold the text being edited for the duration of the editing session. At the end of the session, the buffer is usually copied to the permanent storage file where it normally resides when no edits are being performed on the file. It is easy, particularly when performing complex edits, to make a mistake that can have disastrous effects on the buffer file. These mistakes are usually corrected without difficulty by using the **undo** command. However, a mistake is sometimes not discovered until it is no longer recoverable by **undo**, making it necessary to abort the edit and start over. Much of the lost work can be recovered if the user was cautious and saved the buffer file in permanent storage periodically during the editing session, especially before any complicated changes.

Prudent computer users usually update the stored version of the file they are editing several times during the editing session, especially when the editing session is long and tedious. By keeping the permanent file up to date, they cannot lose more than a few minutes' work, at most, if an operator error, system crash, power failure, or other interruption unexpectedly stops the editing session or damages the buffer file beyond convenient repair. It is comforting to know that HP-UX has extensive protective features that help prevent loss of data due to system crashes and power failures. However, these protection mechanisms are useless against operator errors. You can easily update the permanent storage file with the current contents of the buffer by using the *vi* command:

```
:w RETURN
```

This command copies the current workfile onto permanent disc storage in the filename specified when *vi* was invoked. It does not change your current location in the file being edited, and returns you to *vi* so you can conveniently continue with your next command as soon as the write operation is complete. This simple operation can be especially useful when you are about to try a complex search-and-replace operation and want to make sure you don't lose the work you have already done on the file.

Repeating the **:w** command every few minutes keeps your permanent file up to date. This means that you have little opportunity to destroy your work if you make an incorrect keyboard entry in a moment of inattention. The risk of lost work is particulary important to fast touch typists who occasionally forget to enter input mode before typing new text, especially when the text happens to coincide with a command sequence that destroys part of the file beyond easy recovery.

Using the Write Command

The write command:

:w *filename* RETURN

can be used at any time to store the current buffer file in a file identified by *filename* (include the full or relative directory pathname if the file is not being stored in the current directory). This is a convenient way to store slightly different versions of the same file in more than one location. This version of the write command should be used when you want to prevent accidentally overwriting an existing file.

Overwriting Files

You can overwrite the contents of an existing file with the contents of the buffer file by using the overwrite command:

:w! *existing_filename* RETURN

If you use **:w!** and the file does not exist, the command is treated the same as **:w**. Remember that no space is allowed between the **w** and the **!** when writing the buffer to a file.

The exclamation point (usually pronounced "bang" by HP-UX users) tells *vi* one of two things:

- You know the file exists and you want to overwrite the old file (destroying its previous contents), or
- You want the file written to that filename, even if an existing file having the same name would be destroyed by the write operation.

If you use the **:w** command and a file having the specified filename already exists, *vi* displays the warning message:

"<*filename*>" File exists - use "w! <*filename*>" to overwrite

You can continue editing, select a different filename, or use the **:w!** command (no space before the **!** to overwrite the existing file. You can also abort the edit by using the command:

:q! RETURN

When you have finished editing and are ready to store the file on the previously specified disc file, send a write (**:w** or **:w!**) or exit (**ZZ**) command to *vi*. The write command copies the buffer file to the specified (or current if not specified) filename then returns to *vi*. To end the session after a write command, use the quit command. **ZZ** copies the buffer file to the current *filename*, destroys the temporary buffer file, then terminates *vi* (equivalent to **:wq**).

When invoking *vi* with an existing filename, the file must reside in the current working directory. If the file is located in a different directory, the appropriate directory pathname must also be provided with the filename. Any legitimate pathname can be used including the substitution characters **.** and **..** for the current directory and parent of the current directory, respectively.

Creating Multiple Versions of a File

Occasionally, you may need to create several files that are similar in some respects, but different in others. Or, you may want to split a long file into several smaller files. The **:w** command can be used to accomplish this with little difficulty by copying all or part of the file being edited to another file or files. The use of this and other commands to copy or split files and perform other useful tasks are described in greater detail in the File Manipulation Techniques chapter.

After using the write command, you can end the *vi* session or continue editing until you are ready to record another file or end the session. Whenever you choose a normal termination by pressing **ZZ**, the current buffer file, as edited, is rewritten over the file originally specified when *vi* was invoked.

Note Concerning :w! and :w !

The previous paragraphs describe the use of **:w!** without any space between the **w** and **!** to overwrite existing files. This command is radically different from the **:w !** with a space between **w** and **!** which writes (all or part of) the buffer out as standard input to an HP-UX command. This second form (**:w !**) is discussed near the end of Chapter 11 which deals with shell operations.

Beware the Wiles of ZZ

For most normal, simple editing where you open a file, alter it, then store it back under its original name, you can indiscriminately terminate with either **:wq** or **ZZ** and obtain the same, identical, and desired effect. However, if you are a more advanced user or performing more specialized tasks, you may encounter some surprising and unwanted effects if you indiscriminately use **ZZ** after certain types of **:w** operations.

When a new file is opened, *vi*/*ex* sets a "modified-file" flag when the first change is made to the buffer file. Whenever a **:w** or **:w!** command is used to write the entire buffer file to permanent storage, the flag is cleared, whether the buffer was written to the original file or to some other file (if only part of the file is written, the flag remains unchanged).

On the other hand, when **ZZ** is used to terminate an editing session, it examines the modified-file flag. If the flag is not set, it exits the session without writing the buffer file back to its original filename in the file system. This means that if you open file A for editing, modify it and write the entire buffer to file B, then use **ZZ** to terminate, file A remains unchanged (since no changes were made after writing to file B) and file B contains the modified form of the original file A. This may or may not be what you want, depending on whether you need to update the original text file.

If you want to update the original file so that it is identical to the buffer before you leave the editor and you have written the entire buffer to another file by using **:w** or **:w!**, you must force the file to be written back to the original file. As you might expect, there are multiple ways to accomplish this. Here are two:

- Use **:wq** without specifying a file name, or **:w!** followed by **:q** if the file is read only or has been assigned "edited" status by *vi*. Be sure that the name of the current file is the same as the original. If you are not sure, use CTRL-G to list the information about the current file at the bottom of the screen or type:

  :file RETURN

  to get the same information. If the filename is not correct, use the **:file** *filename* command described in Chapter 10 to change it back, or simply use the **:w!** *filename* to force a write to the existing original file.

- Perform a harmless modification such as **ddP** or **xP** to set the modified file flag, then use **ZZ** as usual. For most users, this method is probably easier and less susceptible to error.

Writing to an HP-UX Command

The **:w** command can also be used to send all or part of the buffer file to any HP-UX command for processing. This is helpful when you want to print a sample section of text on the system printer, change format before storing in a file, converting control characters to printable visible character sequences, and so forth. Procedures and several examples are explained in detail near the end of Chapter 11.

Using Saved Text

Once a new file contains text and has been written to permanent storage, it is treated as an existing file from that time on. Most of the topics discussed in this tutorial can be tried on a text file that contains two or more paragraphs. Use the previous discussion to create a sample text file containing two or more paragraphs, then type **ZZ** to store the file. Restart the editor, specifying the stored file name, and wait until the text appears on the terminal display screen.

For example, if you start by using the command:

```
vi junk [RETURN]
```

add several lines of text, then type:

```
ZZ
```

to terminate the session, a new file named *junk* is created in the current directory that contains the new text you just typed. You can now edit the file by using the same command you used before:

```
vi junk [RETURN]
```

but instead of the `[New File]` message at the bottom of the screen, you see `"junk"` followed by the number of lines and characters in the file. For example:

```
"junk" 21 lines, 382 characters
```

The cursor is located at the beginning of the file, and *vi* is waiting for your first command. You are ready to proceed as described in later chapters.

Recovering from a System Crash

HP-UX system crashes, other than shutdowns caused by power failures, are usually rare. However utility company power failures do happen, and there is always the possibility that someone may accidentally knock a power cord loose in a moment of carelessnes or inattention, so such events must be provided for and coped with. Great care was taken when designing HP-UX to provide a high immunity to problems related to power failures so there is little need for concern other than knowing how to recover when power is restored.

Power-Failure and System Crash Protection

Most system disk drives used with HP-UX have automatic power-fail detection. If a power failure occurs, they immediately write any data being held by the drive controller onto the correct disk location then withdraw the heads to prevent a head crash. However, the system computer may be holding additional data in memory buffers that have not been emptied to the disk drive controller. During of a power failure, the disk controller does not accept attempted output from the computer, because it is busy shutting down the disk drive.

To protect against the loss of data being held in memory by the computer when power fails, Series 300 HP-UX systems include a System Administration command named *syncer*. *syncer* periodically (every 30 seconds unless specified otherwise) and automatically empties output buffers to system disk storage so that little information is being held in user buffers. This means that if *syncer* is running on your system, you will not lose more than the last 30 seconds of work if the power goes off. Contact your System Administrator to find out if *syncer* is being used on your system. Series 500 systems use a different system architecture, so *syncer* is not needed, and is therefore not included in Series 500 HP-UX software.

Recovery Procedure

When a system shutdown occurs, the original file being edited is still in its original location and contains the same data it contained at the start of the session or after the last write command that updated it from the buffer file. The preserved buffer file, on the other hand, contains the buffer file as it was following the last buffer update by *vi*. If *syncer* was in operation, the computer's buffer memory was dumped to the disc soon before the crash. If *syncer* was not in operation, all the data being held in the computer's internal buffer memory since the last disk write operation from the buffer was probably lost, making the buffer file less up to date.

Recovering the buffer file from the previous session is easy because *vi* provides a convenient means for recovering from a power failures and system shutdowns. The recovery procedure described below conveniently resumes your interrupted session. It is important to note, however, that this protection against power failure does not reduce the value of periodically updating permanent storage by writing the buffer to a permanent file as described earlier in this chapter.

When a power failure or other condition causes the system to shut down or crash, *vi* buffer files are maintained in the temporary storage directory */tmp*. When the system is restarted, you will probably receive mail from the system indicating that your *vi*/*ex* buffer file has been preserved along with instructions on how to recover it. To continue editing that file, use the *cd* command to change your current working directory to the same directory you were in during the edit, then invoke the *vi* command exactly as you did for the previous interrupted session **except** using the **−r** option as follows:

```
vi -r filename RETURN
```

The file will be restored to the terminal screen, and will contain exactly the data it contained at the time the disk drive shut down or the system crashed. You can then use the **:w** or **ZZ** command to update the original file, or you can continue editing; whichever you prefer. In the vast majority of cases, your file will be missing the last line typed or the last few keystrokes, but rarely will there be extensive damage unless the editor was in the midst of a complex undo operation or some similar condition when the power failure occurred.

If you are using another editor such as *view*, *ex*, or *edit*, use the same command form; just use the appropriate command:

```
view -r filename RETURN
```

```
ex -r filename RETURN
```

or

```
edit -r filename RETURN
```

Do not wait several days before recovering the preserved file. Editor buffer files are stored in directory */tmp*. Prudent System Administrators usually discard files in */temp* on a regular basis to conserve disk space and keep the file system clean. If you cannot be available when the system is restored to operation, it is recommended that you arrange for someone to copy the file into another directory where you can recover it later. You can usually identify your buffer file by its name. Use the HP-UX command:

```
ll /tmp RETURN
```

The filename listed in the right-hand column will start with **Ex** followed by a (usually 5-digit) process ID number. The file owner column normally contains your login ID unless you have changed to another user name while logged in. If you need assistance, contact your System Administrator.

Basic Editing: Cursor and Display Control

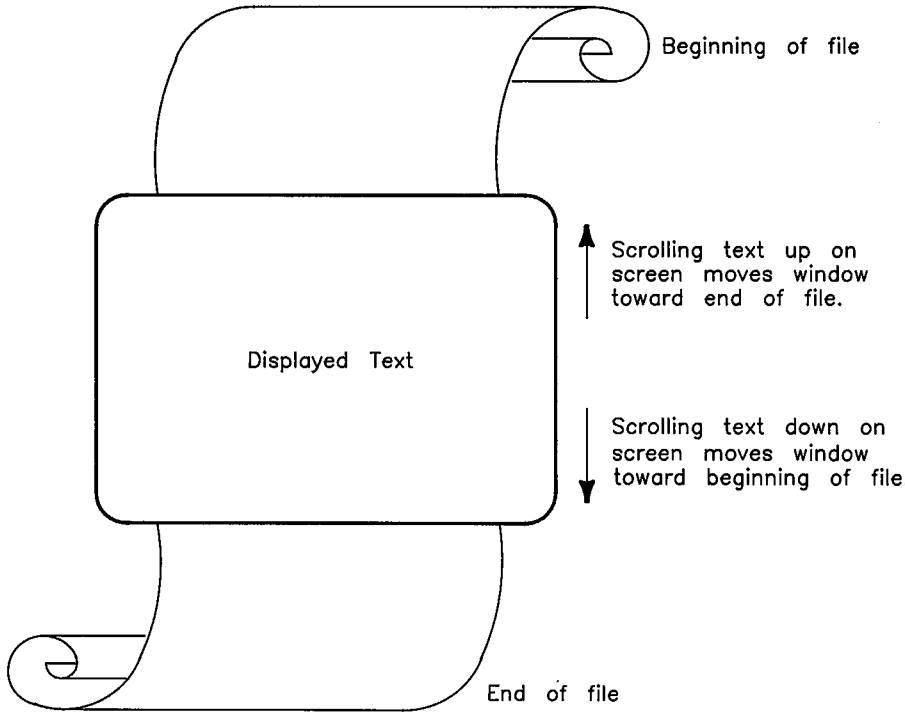
3

Once you have an existing file to edit, you are ready to alter existing text. However, editing is much easier when you know how to control cursor movement, scrolling, and other CRT display features provided by *vi*. This chapter addresses two general topics:

- The beginning sections of this chapter explain how to use common screen control operations for moving quickly to a location of interest before making desired changes in text.
- Later sections describe cursor movement commands that are commonly used to alter text objects and blocks of text and accomplish other useful tasks.

Editor Window Operation

Most terminals display about 20 to 24 lines of text (excluding the command/message line at the bottom) with up to 80 characters horizontally in each line. These lines are your access window into the file you are editing. When *vi* starts up, the window displays the first part of the file with line 1 of the window being in the same position as line 1 of the file as shown in the figure below.



CRT Display Provides a Viewing Window into a File

You can use cursor control keys to move the cursor to any position in the visible screen area, and use scrolling control keys to move the text up or down on the screen. As text scrolls up, the viewing window moves toward the end of the file. To get a good feel for using *vi*'s screen control features, you need a file that is much longer than two full screens (about 50 lines).

Finding a Sample File to Edit

Let's be creatively lazy. Rather than typing a long file just to try a few cursor control exercises, let's use a text file that already exists. An easy place to find relatively large text files to experiment with is the on-line manual page file set. Manual page files are part of the HP-UX operating system and should not be modified by anyone who is not properly authorized to do so. However, since only the system administrator (or other super-user) can write to these files but anyone can read them, they make good source material for experimenting. **Do not try the following exercises if you are logged in as super-user or have super-user capability!**

Be sure you are logged in as an ordinary user in your home account. Use your home directory, or change to some other directory that you own and have write permission in. First we must find the manual page for the *vi* command. Manual pages are stored in a directory under */usr/man* To determine which directory contains the command, use the HP-UX command (not available on AXE):

```
whereis vi RETURN
```

You should get a response similar to the following:

```
vi: /usr/bin/vi /usr/man/man1/vi.1
```

or

```
vi: /usr/bin/vi /usr/man/man1.Z/vi.1
```

This command tells you where the *vi* files are stored. */usr/bin/vi* is the *vi* program itself, while directory *usr/man/man1* contains the *vi* manual page in normal text form and *usr/man/man1.Z* contains the *vi* manual page in compressed text form.

If you are using AXE, execute the HP-UX command:

```
ls /usr/man RETURN
```

The file listing should contain a directory named *cat1* and/or a directory named *cat1.Z*. If directory *cat1.Z* is present, it probably contains the file *vi.1* in compressed form. To verify, use the command:

```
ls /usr/man/cat1.Z/vi* RETURN
```

for compressed pages, or

```
ls /usr/man/cat1/vi* RETURN
```

if the compressed pages directory is not present, and look for a file named *vi.1*.

Copying an Uncompressed Manual Page

If the *vi.1* file is in an uncompressed directory (*/usr/man/man1* or */usr/man/cat1*), copy the file into the current directory by executing one of the following commands:

On an ordinary HP-UX system,

```
cp /usr/man/man1/vi* ./junk RETURN
```

places a copy of the unformatted on-line manual page file */usr/man/man1* into a new file named *junk* in the current working directory.

If you are using an AXE (Applications eXecution Environment) system, you will have to copy the formatted page as follows:

```
cp /usr/man/cat1/vi* ./junk RETURN
```

This file is formatted (by *nroff*) and may or may not be largely unreadable because of character and word highlighting sequences. However, it can still be used for experimenting with cursor control.

Uncompressing and Copying a Compressed Manual Page

If file *vi.1* is in directory *man1.Z* (or *cat1.Z* on AXE), the files are compressed. To copy the file into a form you can use, execute the command:

```
uncompress -c </usr/man/man1.Z/vi.1 >./junk RETURN
```

on standard HP-UX, or

```
uncompress -c </usr/man/cat1.Z/vi.1 >./junk RETURN
```

on an AXE system. The file *junk* is now ready for use if no errors occur.

Note

Experienced users may wonder why the *zcat* command is not used in this example. *uncompress* and *zcat* both expect a **.Z** suffix on the filename. Compressed manual pages use the .Z on the directory name instead, thus requiring use of the **−c** option to recover any compressed manual page files.

Using the File

Now start an editing session by typing:

```
vi junk RETURN
```

Don't panic when the screen fills with strange character sequences mixed with text. File *man1/vi.1* or *man1.Z/vi.1* contains the source text used to typeset the printed *HP-UX Reference* pages through a *troff* formatter program. This file is also used by the built-in HP-UX *nroff* program used by the *man* command when printing a formatted manual page on the display screen. If you are using an AXE system, your file will look different because the *vi.1* file in directory *cat1* or *cat1.Z* is produced by *nroff* from the *man1* version prior to producing the AXE software package. This formatted version may contain some rather bizarre character strings that are used for terminal highlighting. That's not important – we just need a nice long file to play with while learning to use the editor.

Notice that the first line on your terminal screen looks like the beginning of the file. The next several topics in this chapter help you quickly learn how to move around in the file.

Determining File Size

When *vi* first displays the file on the screen, it prints some information about the file on the bottom line, including the number of lines and characters in the file. You can recover this and other information about the file during an edit by using the CTRL-G or **:f** command discussed in the next section.

Positioning the Cursor on the Current Line

The cursor is easily moved back and forth within the current line while the editor is in command mode by using the space bar (or **l** key) and [BACK SPACE] (or **h**) key (use of **h** and **l** is explained next after the arrow keys topic). Use the following to conveniently move to specific locations at the beginning or end of the line:

Beginning/End-of-Line Cursor Positioning Commands

Command	New Cursor Position
0 or |	Move cursor to column 0 (left edge) of screen on current line.
n|	Move cursor to column *n* on current line.
^	Move cursor to first non-blank character on current line.
$	Move cursor to last character (blank or otherwise) on current line.

Arrow Keys

If you are accustomed to using arrow (up, down, left, right, home) keys to control your terminal display, *vi* provides some improvements. While you may find them a bit clumsy at first, after some practice you will learn to appreciate the many short-cuts that help you control the display without moving your fingers all over the keyboard.

Vi works correctly with the arrow keys on many terminals, but not all (how well they work depends on your $TERM setting and how the terminal characteristics have been defined in the *terminfo* database, among other things). Use of arrow keys is discouraged, partly because they do not always work correctly, but also because using the "home row" keys **h** (or [BACK SPACE]), **j**, **k**, and **l** (or space bar) is faster and much more convenient for users such as touch typists who prefer to keep their fingers on the home row. However, for those less expert users who feel more comfortable with arrow keys, most HP terminals handle arrow keys without difficulty. If you are a frequent user of *vi* and take the trouble to learn to use the home row keys, you will likely find the effort rewarding once you become accustomed to them.

Positioning the Cursor on the Screen

Cursor screen-positioning commands are used to manipulate the cursor's current location on the screen and to control the position of the screen window in the file being edited. The cursor control commands shown here are for general moving around in the file. **Other methods** for moving the cursor to a specific position in the text file by searching for text patterns, character location on current line, word, paragraph, or section beginning or end, and such are covered later in this chapter as well as elsewhere in the manual as various related topics are introduced.

Here are the most commonly used cursor positioning commands. *n* represents a numerical value that is typed before the command. [RETURN] is not used after these commands unless specifically shown. Some of these commands can also be used in conjunction with editing commands when deleting or changing several lines or larger blocks of text

Screen/File Cursor Movement Commands

Command	Cursor Motion
h	(or [BACK SPACE]) Move cursor left one character
*n***h**	(or [BACK SPACE]) Move cursor left *n* characters
j	Move cursor down one line; stop at bottom of screen and scroll if necessary.
*n***j**	Move cursor down *n* lines; stop at bottom of screen and scroll if necessary.
k	Move cursor up one line; stop at top of screen and scroll if necessary.
*n***k**	Move cursor up *n* lines; stop at top of screen and scroll if necessary.
l	(or Space Bar) Move cursor right one character
*n***l**	(or Space Bar) Move cursor right *n* characters
[RETURN]	Move cursor to beginning[1] of next line.
n[RETURN]	Move cursor to beginning[1] of the *n*th line after current line.
+ or *n*+	Same as [RETURN] and *n*[RETURN]
− or *n*−	Same as [RETURN] and *n*[RETURN] except that movement is toward beginning of file. For example, − moves to the first visible character in the preceding line.

[1] Beginning of line in this context is the first visible character in the line or end-of-line, whichever occurs first.

Screen/File Cursor Movement Commands (continued)

Command	Cursor Motion
G	Move cursor to first visible character in last line of file.
1G	Move cursor to first visible character in first line of file.
*n*G	Move cursor to first visible character in line *n* of file.
CTRL-G or :f RETURN	Display line number of cursor line and other information about the file on the bottom (command) line of the terminal display.
0 or |	Move cursor to absolute beginning of line (left column of screen).
^	Move cursor to first visible character in current line.
$	Move cursor to last character in current line.
n|	Move cursor to column *n* in current line.
H	Move cursor to beginning[1] of line at top of screen.
*n*H	Move cursor to beginning[1] of *n*th line from top of screen.
M	Move cursor to beginning[1] of line at middle of screen.
L	Move cursor to beginning[1] of line at bottom of screen.
*n*L	Move cursor to beginning[1] of *n*th line from bottom of screen.

Note

When using **j**, **k**, or up/down arrow keys to move the cursor up or down, if a vertical cursor movement would place the cursor beyond the last character of the new cursor line, the cursor is placed at the end of the new line. However, *vi* remembers the column where the cursor began its vertical movements, so if you move the cursor to an equal or longer line than the original cursor line, the cursor is placed it its original column on the terminal screen.

The exception to this rule is when a **$** command moves the cursor to end of line. Any subsequent **j**, **k**, or up/down arrow commands move the cursor to the end of the target new cursor line, independent of line length.

[1] Beginning of line in this context is the first visible character in the line or end-of-line, whichever occurs first. For example, if a line contains five blanks (spaces), the cursor is placed at the last blank character in the line. If the line contains no characters, the cursor is placed at the left margin of the display screen.

What is that Beep I Hear?

In the preceding list of cursor positioning commands, *n* is any positive integer numerical value (obviously, it must not be preceded by + or −), but it cannot exceed the number of lines between the current line and the end or beginning of file if movement is vertical. For example, if you are editing a 100-line file and the cursor is on line 55, you cannot use the command:

58 RETURN or 58+

because 55 + 58 is more than 100 lines. The same is true if you attempt to use the command:

62-

because you cannot back up beyond line 1. Whenever you try to move beyond beginning or end of file, *vi* responds with a beep and ignores the command.

On the other hand, if you are moving horizontally, *vi* always tries to meet your requirements. For example if the cursor is located in column 45 of the current line and the line contains 68 characters, the commands:

100 l or 100 (spacebar)

and

100 h or 100 BACK SPACE

move the cursor to the last character or first character respectively (including tabs and spaces) in the current line without complaint.

Exercise Time

Using the *junk* file on your screen, try a few combinations of these commands until you feel comfortable with their behavior. Here are some suggestions. Try others on your own:

Move cursor to the end of the 25th line following the current line (**$ 25 j** or **25 j $**).
Now move to the first visible character of the 10th line preceding (**10 −**).

Go to the end of the last line in the file (**G $**). Why doesn't **$ G** give the same result when the last line has more than one visible character?

Scrolling Text

Once you have reasonable mastery of cursor movement, you are ready to learn scrolling techniques. Here are the most common scrolling commands ([CTRL] and the following character are pressed simultaneously; *n* is a numeric value that is typed before the command):

Command	Resulting Text Motion
[CTRL]-[B]	Scroll backward to the previous screen.
n[CTRL]-[B]	Scroll backward to the *n*th previous screen.
[CTRL]-[U]	Scroll backward one half screen.
n[CTRL]-[U]	Set half-screen scroll to *n* lines then scroll backward one-half screen.
[CTRL]-[D]	Scroll forward one half screen.
n[CTRL]-[D]	Set half-screen scroll to *n* lines then scroll forward one-half screen.
[CTRL]-[F]	Scroll forward to the next following screen.
n[CTRL]-[F]	Scroll forward to the *n*th following screen.
[CTRL]-[Y]	Scroll backward one line.
n[CTRL]-[Y]	Scroll backward *n* lines (cursor movement explained below).
[CTRL]-[E]	Scroll forward one line.
n[CTRL]-[E]	Scroll forward *n* lines (cursor movement explained below).

These commands are simple and straight-forward, especially if you have mastered the cursor control commands. The most unexpected characteristic in these commands is the numbered half-screen scroll.

Half-Screen Scrolls

The numbered half-screen scroll sets the number of lines to be scrolled to *n*, then scrolls up or down that many lines. From that time on, any half-screen scrolls, whether up or down, will use the same number of lines (unless you specify a new value for *n*). The default (no *n* specified) is about 10-12 lines, but you can select any number (within reason). However, values greater than 20 would probably not be very useful.

Cursor Movement During Scrolling

If scrolling moves the current (cursor) line beyond the screen boundary, *vi* usually positions the cursor at the first visible character or end-of-line (whichever occurs first) at the top (forward scroll) or bottom (backward scroll) of the display screen.

If you are using CTRL-Y or CTRL-E, *vi* leaves the cursor at its current location **in the file** until the cursor is forced off the screen at which time it remains on the top or bottom line and usually moves to the first visible character in the line.

Positioning the Cursor Line in the Display Window

Sometimes you may want to move the current cursor line to a different position in the display window. One example is when forward scrolling leaves the cursor at the last line in the file and at the bottom of the screen. You can easily move the cursor line to the top, middle, or bottom of the screen as follows:

Commands to Reposition Cursor Line on Screen

Command	Resulting Text Position
z RETURN	Move cursor line to the top of the screen. Scroll surrounding text accordingly. z+ cannot be used for the same purpose.
z .	Move cursor line to the middle of the screen. Scroll surrounding text accordingly.
z –	Move cursor line to the bottom of the screen. Scroll surrounding text accordingly.

Where Am I in the File?

Vi is a visually-oriented rather than a line-oriented editor, so you usually have no need to know line numbers. However, there are occasions when it is helpful to know the current line number. A common instance is the need to specify a line number or range of line numbers when using an escape to the *ex* editor for global search-and-replace operations.

The **G** or *n***G** command is used to move to a specific line in the file. To determine the number of the current cursor line at any time, place the editor in command mode, then use the command:

CTRL-G

or

:f RETURN

Vi lists several items of information at the bottom of the screen on the command line using a format similar to the following:

```
"junkfile" line 788 of 870 --90%--
```

or

```
"junkfile" [Modified] line 788 of 870 --90%--
```

The first example shows the cursor is currently located on line 788 in an 870-line file named *junkfile*, and about 90% of the file precedes the current cursor line (the file has not been modified). The second example is the same line in the same file, but the file has been modified since the editing session began.

Note

vi can be configured to display line numbers along the left side of the display screen (they are not added to the file being edited) with some loss of available text display space. To display line numbers, use the **:set number** command discussed in the chapter entitled Configuring the Vi/Ex Editor.

Practice Time

Try some scrolling sequences on the *junk* file on your screen until you feel comfortable with their behavior.

But I Can't Scroll Forward Using CTRL-F

Sometimes CTRL F does nothing. This is not a bug in *vi*. CTRL-F is the ASCII ACK control character used in ENQ/ACK data communications protocol, and it gets discarded by any terminal or software interface that is set for ENQ/ACK protocol. If your terminal does not use or need ENQ/ACK protocol, you can correct the problem by disabling ENQ/ACK protocol with the following command:

```
stty -ienqak echoe
```
from HP-UX, or

```
:!stty -ienqak echoe
```
from *vi*.

If your terminal uses ENQ/ACK protocol, you cannot use the CTRL-F scrolling feature. If you do not use ENQ/ACK protocol on your terminal, you can add the HP-UX version of the *stty* command above to your *$HOME/.profile* file, thus eliminating having to do this each time you use *vi*.

Searching Through a File for a Pattern

An important feature in text editors is the ability to search for a pattern of characters in a file and display the surrounding text on the terminal screen. *vi* has several commands that can accomplish this task, depending on the situation. The most common technique is through use of the / and ? commands.

The / and ? commands are used for mainly two purposes:

- Positioning the text being searched for within the range of the current viewing window as described in this section.
- Establishing boundaries for text-object modifications as described later in this chapter.

When you type / followed by a text pattern or regular expression (regular expressions are treated in greater detail in the tutorial on regular expressions earlier in this volume), *vi* searches the file for the first occurrence of the pattern beginning at the current cursor location, then displays the surrounding text if the pattern is found. The search is conducted in the forward direction. If you are not at the beginning of the file and end-of-file is encountered before the pattern is matched, the search wraps to beginning of file[1] and continues until the pattern is found or the cursor location is again reached, meaning that the pattern does not exist in the file.

The **?** command works the same way except that the search is conducted in the reverse direction (backwards in the file). If the pattern does not appear between the cursor line and beginning of file, the search wraps to end of file[1] and continues as before.

Repeating the Search

Sometimes an expression appears several times in a file. You can search for the first occurrence by specifying the pattern. To find additional occurrences of the pattern, it is not necessary to repeat the command. Simply use **n** to repeat the search in the same direction, or **N** to repeat the search in the opposite direction.

Aborting a Long or Incorrect Search

Occasionally, especially when editing extremely large files, you may enter a search string that does not exist or make a typographical error when entering the search string. To abort the search, simply press the [BREAK] key. The editor returns a message on the bottom line of the CRT display:

```
Interrupt
```

You can then proceed with a corrected search command or do something else.

[1] End-of file wrap-around requires that the **wrapscan** option be enabled, which is the normal default condition. Refer to the chapter entitled Configuring the Vi/Ex Editor for more information.

In case of Difficulty

While *vi* rarely misbehaves, it can present some perplexing problems on occasion. Here are a few CRT display problems you may encounter along with suggested solutions.

Restoring a Garbled Display

Occasionally, you may be in the middle of an editing session when the text on your display becomes garbled. This can happen when noise on the line between terminal and computer causes characters to be incorrectly received, someone writes a message on your terminal (sometimes as an annoying prank), you try to display a non-ASCII file using a command such as *more*, *head* or *tail*, or something else causes the display to be altered so that you cannot use *vi* correctly. *Vi* provides an easy solution. Press:

CTRL-L simultaneously

to tell *vi* to redraw the screen. This usually solves the problem. If the problem persists or data is repeatedly incorrect, contact your system administrator for help in diagnosing the cause.

What If Screen Behavior Becomes Strange?

HP-UX and *vi* use the lowest possible number of characters to produce new text on the user's terminal display. Sometimes the system uses tab characters to accomplish this objective. As a result, you may occasionally be moving the cursor along a line while *vi* is in command mode only to discover that the display text starts changing as the cursor moves or blocks of text suddenly relocate on the screen. This is usually caused by the tab stops being altered for any of various reasons but most commonly by embedded control characters in text that cause the terminal to interpret text characters as screen configuration commands. To correct the problem, execute the HP-UX command:

`:!tabs` RETURN from *vi*, or

`tabs` RETURN from HP-UX

then continue. If the problem persists, clear all the tab stops on the terminal by using the AIDS keys or by placing your terminal in local (REMOTE MODE off) then pressing ESC 3 (standard for HP terminals; varies with other brands – refer to terminal operating manual for correct sequence).

Place the terminal back in remote (REMOTE MODE on) then execute the *tabs* command again as follows:

tabs [RETURN] from HP-UX, or

:!tabs [RETURN] from *vi*

then resume using *vi*. This sequence of operations should clear up most problems. If this does not work, you can write the file to permanent storage, log off, cycle the terminal power switch off then on again, then log in again and restart the editor.

This is a good time to talk about manners. The multi-user and networking capabilities in HP-UX make it a very useful tool. At the same time, power given to a user demands responsible use of that power. Don't use the system to annoy others who may not appreciate your humor, especially when they are in the middle of a complex operation. Treat others as you would want them to treat you if you were in a similar situation.

Conflicts Between Commands and Terminal Protocol

Some *vi* scrolling commands and any other commands that use [CTRL] key sequences may be identical to characters used in terminal data communications protocol. Thus, if you are using enq/ack protocol for the terminal connection and you send a [CTRL]-**F** (an **ack** character) to *vi*, it is consumed by the terminal interface and does not reach its intended destination. This leads to the problem with [CTRL]-[F] not working as described earlier in this chapter. Similar problems occur when you use [CTRL]-[D] and it is being used as an end-of-file character. These problems are discussed at length in the chapter entitled Configuring the Vi/Ex Editor.

Characters that most commonly produce difficulties include DC1, DC3, ENQ, and ACK in the handshaking group, DEL (sometimes called DLE), and ETX (^C) sometimes used as an interrupt character, and the quit character configured for your terminal by the *stty* command in your login script file.

Positioning the Cursor in the File

Discussions of cursor movement in the preceding parts of this chapter are related to positioning the cursor on the CRT display screen and moving the screen display window around in the file. The topics discussed in the remainder of this chapter are focused on moving the cursor to specific locations in a file, particularly with regard to how such movements relate to text modification commands discussed in Chapter 4 and elsewhere in this tutorial.

Moving to a Specific Column Number

In some situations, you may want to move quickly to a particular column on the current line. Two commands move the cursor to the left end of the line:

Command	Result
^	moves the cursor to the first visible text character in the line.
| or **0**	moves the cursor to the extreme left column on the current line.
n|	moves the cursor to column *n* in the line.
*n***SPACE BAR**	moves the cursor *n* spaces to the right from current position.
n[BACK SPACE]	moves the cursor *n* spaces to the left from current position.

Text Objects

A text object, in *vi* parlance, is an arbitrary collection of text between the current cursor position and some other user-defined location in the file. *vi* provides various cursor move commands for identifying words, lines, sentences, paragraphs, sections, or other text blocks as the text object to be acted upon by an editing operation. In general, the boundaries chosen by *vi* when identifying an object such as a word, sentence, or paragraph closely resemble the interpretation most people would use given the same situation. However, there are some important (and quite useful) differences that are discussed in detail during the next several topics.

Using the delete (**d**), change (**c**), and yank (**y**) editor commands in conjunction with cursor move commands that define the text object being acted upon provides a useful means for deleting, altering, copying or moving small or large blocks of text anywhere in the file being edited. But as you might expect, getting full value from these capabilities requires that you be aware of their availability and how they can be used.

The Find Commands: f, F, t, and T

vi provides four cursor move commands for searching forward or backwards in the current line for the next or *n*th occurrence of a given character. They do not search beyond beginning- or end-of-line as the case may be. The commands are as follows:

Find Commands: Search for Character within Current Line

Command	Action
f*c*	Forward to next occurrence of character *c*.
*n*f*c*	Forward to *n*th occurrence of character *c*.
F*c*	Backwards to next occurrence of character *c*.
*n*F*c*	Backwards to *n*th occurrence of character *c*.
t*c*	Forward to character before next occurrence of character *c*.
*n*t*c*	Forward to character before *n*th occurrence of character *c*.
T*c*	Backwards to character after next occurrence of character *c*.
*n*T*c*	Backwards to character after *n*th occurrence of character *c*.
;	To next occurrence of character *c* in same direction as previous search.
n;	To *n*th occurrence of character *c* in same direction as previous search.
,	To next occurrence of character *c* in opposite direction from previous search.
n,	To *n*th occurrence of character *c* in opposite direction from previous search.

where search directions and character positions are:

- **forward**: Toward end of file from current position.
- **backwards**: Toward beginning of file from current position.
- **after**: Adjacent to target character, but toward end of file.
- **before**: Adjacent to target character, but toward beginning of file.

These commands apply only to the current line, and cannot be used to move the cursor to another line.

Examples

Consider the following sentence and assume that the cursor is located under the **y** in **Wiggly**:

```
Willie the Wiggly Worm went to Washington to wander in wonder.
```

Here is a list of commands and the associated cursor movement:

Command	Cursor Location after Move
fW	**W** in **Worm**
2fW	**W** in **Washington**
2fw	**w** in **wonder**
3fW	No movement because character does not exist.
tW	Blank space before **Worm**
2tW	Blank space before **Washington**
2tw	Blank space before **wonder**
3tW	No movement because character does not exist.
FW	**W** in **Wiggly**
2FW	**W** in **Willie**
2Fw	No movement because character does not exist.
3FW	No movement because character does not exist.
TW	**i** in **Wiggly**
2TW	First **i** in **Willie**
2Tw	No movement because character does not exist.
3TW	No movement because character does not exist.

The Word Commands: w, W, e, E, b, and B

vi provides four cursor move commands for searching forward or backwards in the file by words: **w** or **W** for forward moves, and **b** or **B** for moving backwards in the file. Word boundaries are defined as the imaginary zero-width space at the beginning of the next word. Thus any given word includes the white space between it and the next word, if any exists. Moving by words is unlimited within the file and is not restricted to the current line.

vi interprets word boundaries in two ways. You must define which interpretation is to be used as part of each command. Using a lowercase word move command (**w** or **b**) treats any non-alphanumeric character except underscore (_) as part of the next word. An uppercase word move command (**W** or **B**), on the other hand, uses white space (space, tab, or new-line character) as a word separator and the next word begins at the next character past the one or more whitespace characters adjacent to the end (forward moves) or beginning (backwards moves) of the current word.

Word Commands: Search for Specified Beginning or End of Word

Command	Action
w	Forward to next beginning of word or first non-alphanumeric character.
*n***w**	Forward to *n*th beginning of word/non-alphanumeric character.
W	Forward to next beginning of word; only white space as word separator.
*n***W**	Forward to *n*th beginning of word; only white space as word separator.
e	Forward to next end of word or first non-alphanumeric character.
*n***e**	Forward to *n*th end of word/non-alphanumeric character.
E	Forward to next end of word; only white space as word separator.
*n***E**	Forward to *n*th end of word; only white space as word separator.
b	Backwards to next beginning of word or first non-alphanumeric character.
*n***b**	Backwards to *n*th beginning of word/non-alphanumeric character.
B	Backwards to next beginning of word; only white space as word separator.
*n***B**	Backwards to *n*th beginning of word; only white space as word separator.

Note that all moves are from current cursor position and are independent of where the cursor is within the current word. For example, if the uursor is currently located in the middle of a word, the **b** or **B** command moves the cursor to the beginning of the current word. Likewise, if the cursor is located in the whitespace between two words, a **w** or **W** moves the cursor to the beginning of the adjacent word, not to the beginning of the next word after it.

These commands are not restricted to the current line. The cursor is wrapped to preceding or following lines as necessary in order to meet the specified word count.

Examples of Word Moves

Consider the following line of text:

```
This  line  contains  a  one,annatwo!anna_three&four$five(six)seven  weird  word.
                      ^
```

where the cursor is located at the space character position between the words **contains** and **a** as indicated by the ^ character. To move the cursor from the position indicated to the beginning of the word "weird" requires the command **15w** or **3W**. The command **f2w** would accomplish the same (find the second occurrence of the character "w"; skip the w in "two", stopping at the w in "weird". To move the cursor to the end of the word **seven** (cursor underneath the **n**), use **14e** or **2E**.

When *vi* performs the forward move by 15 words (lowercase command version), text is interpreted as a succession of the following words:

```
a one , annatwo ! anna_three & four $ five ( six ) seven weird
```

counting from the starting cursor position. Using the uppercase form, the same text is interpreted as the following three words:

```
a    one,annatwo!anna_three&four$five(six)seven    weird
```

Note the unvarying treatment of the underscore character in `anna_three`.

Sentence, Paragraph, and Section Commands: ( ) { } [[and]]

vi also recognizes sentence, paragraph, and section boundaries, using a technique similar to that used with words as follows:

- Ends of sentences are detected by the presence of a period (.), question mark (?), or exclamation point (!) followed by **two** or more spaces. A continuum of one or more empty lines (containing no spaces or tabs) is also treated as a separate sentence (a continuum of one or more apparently blank lines, each containing spaces and/or tabs, is treated as part of the preceding sentence).

  To move the cursor to the next adjacent beginning-of-sentence, use) for forward moves or (for backwards moves. Use *n*) or *n*(to move to the *n*th beginning of sentence in the forward or backwards direction, respectively.

- Recognized paragraph boundaries include any paragraph macro defined by the **:set paragraphs** and **:set sections** commands, as well as **empty** lines (blank lines containing no space or tab characters). Default paragraph macros include: **.IP**, **.LP**, **.PP**, **.QP**, **.P**, **.LI**, and **.bp**. Beginning-of-paragraph is defined as the beginning of the first empty line after a paragraph of text or the beginning of a text line that starts with a paragraph or section macro. The macros shown are found in document formatting macro packages such as *mm* or *man* which are both documented in Section 5 of the *HP-UX Reference*. They can be redefined as explained in Chapter 12 by using the **:set paragraphs** command.

  To move the cursor to the next adjacent beginning-of-paragraph, use } for forward moves or { for backwards moves. Use *n*} or *n*{ to move to the *n*th beginning-of-paragraph in the forward or backwards direction, respectively.

- Recognized section boundaries include any section macro defined by the **:set sections** command. Default section macros include: **.NH**, **.SH**, **.H**, and **.HU**. The macros shown are found in document formatting macro packages such as *mm* or *man* which are both documented in Section 5 of the *HP-UX Reference*. They can be redefined as explained in Chapter 12 by using the **:set sections** command.

  To move the cursor to the next adjacent beginning-of-section, use]] for forward moves or [[for backwards moves. Use *n*]] or *n*[[to move to the *n*th beginning-of-section in the forward or backwards direction, respectively.

To add any other macros to the list of recognized paragraph or section macros, use the **:set paragraphs** or **:set sections** command as explained in the chapter entitled Configuring the Vi/Ex Editor.

Using Text Pattern Searches to Define Text Object Boundary

Text pattern searches scan forward or backwards in a text file for a text pattern specified by a **regular expression** included in the command preceding the RETURN. The command format is as follows:

*/regular_expression*RETURN

for forward searches, or

*?regular_expression*RETURN .

for backwards searches.

In its simplest and most commonly used form, *regular_expression* is a simple string of characters identical to the text being searched for. For example,

/thimk RETURN

searches the file for the characters **thimk** either as a stand-alone full word, or as part of a larger word. As you can see, this is a useful means of quickly locating a misspelled word so that it can be corrected.

Other much more elaborate constructions can be used such as:

/the.*wooly.*superstar$RETURN

which searches the file for the first encountered single line, if any, that contains all of the following elements:

Expression	Matching Pattern
`the`	The word **the** anywhere in the line followed by
`.*`	Zero or more arbitrary characters followed by
`wooly`	The word **wooly** followed by
`.*`	Zero or more arbitrary characters followed by
`superstar`	The word **superstar** located at the end of the line.
`$`	End-of-line immediately after the word **superstar**.

Regular expressions provide a powerful means for specifying text patterns used by many HP-UX commands and programs such as *grep*, *awk*, various editors, etc. Regular expressions and their use is the subject of a separate tutorial earlier in this volume. Refer to that presentation for more details as well as examples.

Repeating the Search

You will frequently need to search for a pattern, then repeatedly search for the same pattern in the same file. Once the search pattern has been specified in conjunction with a / or ? command, to repeat the search for the **same** search expression in the **same** direction, press [n]. To repeat the search for the same search expression, but in the opposite direction, press [N].

What is the Exact Boundary of a Text Object

Most users typically have little concern over the exact boundary of a text object because if they guess wrong the mistake is easily fixed. However, for those isolated cases where it is important, here are the general rules for determining where a text object starts and ends.

In general, a text object is bordered by the current cursor position and another location in the file that is determined by a cursor move command or a text pattern search. Its text contents are as follows:

- If the cursor position in the file **precedes** the target location resulting from the move or search, the text object contains all text starting at the cursor position and continuing up to, but not including, the new cursor position character after the move or search is completed.
- If the cursor position in the file **follows** the target location resulting from the move or search, the text object contains all text starting at the new cursor position character after the move or search is completed, and continuing up to, but not including, the original cursor position character.

In simpler terms, the object begins at the earlier cursor position boundary in the file, and ends with the character preceding the second cursor position boundary in the file.

Entire Text Object on Current Line

Here is an example to illustrate. Note the position of the cursor on the second line before the word "particularly".

```
This is example text placed here purely for illustrative
purposes.  It is neither complicated nor particularly long.
                                        ^
```

Let's use the "change text from current postion to first previous beginning of sentence command, **c(**. The cursor immediately moves to the first character in the sentence under the **I** in **It**. A **$** character replaces the last character before the original cursor location (the **r** in **nor**), indicating that all text from the cursor through the position of the **$** symbol will be replaced with whatever text is typed until you press [ESC] (the change command is explained in greater depth in Chapter 4).

```
This is example text placed here purely for illustrative purposes.
It is neither complicated no$ particularly long.
^
```

On the other hand, if you use the **c)** command (change text from current postion to next following end of sentence, notice that the cursor does not move and the **$** symbol is placed at the period's former position at the end of the sentence:

```
This is example text placed here purely for illustrative purposes.
It is neither complicated nor_particularly long$
```

Text Object on Multiple Lines

If the beginning or end of sentence is not on the current line, the text object is handled somewhat differently. For example, in the following text the cursor is located after the first word in the first sentence:

```
This sentence is longer than the one before, so it does not
    ^
all fit on one line.  It also has a second sentence in the
same paragraph.
```

Typing **c)** to change the rest of the sentence causes the text being changed to be removed from the display screen. The paragraph now looks like this with the cursor under the **I** in **It**:

```
ThisIt also has a second sentence in the
    ^
same paragraph.
```

Notice that the whitespace after the period at the end of the sentence is treated as part of the sentence being changed, so it has disappeared. New replacement text is then inserted in front of the cursor character as it is typed until [ESC] is pressed.

Now let's change the second sentence by placing the cursor in front of the last word in the second sentence then use the **c(** command to change to beginning of current sentence:

```
This sentence is longer than the one before, so it does not
all fit on one line.  It also has a second sentence in the
same paragraph.
     ^
```

As before, the text that is to be replaced is removed from the screen. Notice this time that the whitespace after the previous sentence is preserved as well as the character in the starting cursor position. This clearly shows that the whitespace after a sentence is treated as part of the sentence (unless it is an empty line which is treated as a separate sentence in and of itself), and sentence boundaries are interpreted as the beginning of each sentence. As before, new text is inserted in front of the cursor as it is typed until you press [ESC]:

```
This sentence is longer than the one before, so it does not
all fit on one line.  paragraph.
                      ^
```

Basic Editing: Manipulating Text

4

The primary purpose of a text editor is to alter the contents of a text file. Editing operations can be performed on an existing text file or on a new file that is being created in conjunction with the editing session. This chapter explains how to:

- Recover from mistakes ([ESC] and undo).
- Add new text to a file (insert, append, and open).
- Include non-printable ASCII control characters in text.
- Delete text.
- Recover deleted text and move or copy text to other locations.
- Change, replace, or substitute text including changing uppercase to lowercase and vice versa.

Each of these operations can be performed on new as well as existing files. Several examples are used throughout the chapter to introduce each type of operation and demonstrate its use. Learning is much easier if you try them yourself. Before you can try the example exercises, you must terminate any currently active editing session. If you have been using the file *junk* discussed in the previous chapter, quit *vi* by typing:

```
:q! [RETURN]
```

When you get your HP-UX shell prompt (usually a dollar sign), type

```
vi dummy [RETURN]
```

to open a new file for practice. To create practice text, press **A** to enter Append Mode then press [RETURN] a couple of times to create some blank lines. Type the practice line then press [RETURN] another two or three times for more blank lines. Press [ESC] to return to Command Mode.

To move around while editing, use **j** and **k** to move the cursor up or down from line to line, and [RETURN] to move it to the beginning of the next line as needed.

Escaping from the Sand Traps of vi

Like a bad swing in a golf game, a wrong keystroke when using *vi* can put you in a difficult situation one would usually prefer to avoid. The possible errors are numerous, making it impossible to describe every situation and how to get out of it. However, a few simple skills can be easily mastered so you can readily recover and return to the task at hand.

Using the Escape Key

If you find yourself out on a limb, so to speak, press [ESC] a couple of times until you get a beep. The beep acknowledges that you are in command mode so that anything you type will not be placed in the text you are working on. When you get the beep, examine the screen to see if the text near the cursor is as it should be. If the incorrect keystrokes have deleted, inserted, or changed characters that need to be restored, press **u** or **U** (usually the lowercase command is sufficient) to undo the last alteration as explained in the next topic. If you find yourself elsewhere in the file, use the cursor control commands from the preceding chapter to recover your position. To move to the beginning of the file, type **1G**. Use **G** to move to the end of the file. For other moves, use other appropriate commands. The **u** and **U** commands are described next.

Recovering from Mistakes: The Undo Command

Relax. If you make a mistake it is (usually) quite easy to recover; provided, of course, that you do it immediately – not after you have made some other change.

Vi has an "undo" command that reverses the last change made by *vi*, but the command does have some limitations that you should understand (most mistakes are easy to correct, but habitual carelessness can be dangerous). The undo command has two forms that serve two different purposes:

The Undo Commands

Command	Action Taken
u	Undo the most recent text change. If the most recent change was an undo, undo the preceding undo (**u**).
U	Undo all of the changes made to the current line since the cursor was moved to the current line. Not allowed if cursor is moved from current line then returned, with or without a subsequent text change.

The u Command

These commands need additional explanation. The **u** command applies to the most recent text change regardless of present cursor location. It can also be used to undo an immediately preceding undo, provided no other changes have been made since. Thus, if you alter line 50 in a file then move the cursor to line 75 and press **u**, the cursor returns to line 50 and the last change is reversed to its original form (cursor location in the line after the undo operation may or may not reflect the location where the change was made). Pressing **u** again reverses the undo, restoring the original change and leaving the cursor on the changed line. There is no limit on the number of times **u** can be used in succession but it cannot undo more than the last previous change or undo.

The U Command and Examples of use

The **U** command, on the other hand, applies **only** to changes made on the current line while the cursor was on that line, and it can only be used once on that line (unless you make additional changes to the line). If the cursor is moved to another line, the **U** command cannot be used, even if it is returned immediately to the correct line before **U** is attempted. For example, consider the following line of text:

```
This is the original line.
```

Now use two separate insert/append commands to add the two words shown in different locations without allowing the cursor to move to a different line:

```
This is NOT the original UNCHANGED line.
```

Press **U** to restore the original text:

```
This is the original line.
```

Repeat the previous insert/append commands to get the altered sentence:

```
This is NOT the original UNCHANGED line.
```

Now move the cursor to a different line (press RETURN, for example), then move it back to the altered line. Press **U** again. Note that the cursor may move to the beginning of the line, but the text is not changed. Press **U** again. As before, you get no change, but you also get a beep because **U** cannot undo itself, unlike **u**. The altered sentence is still present:

```
This is NOT the original UNCHANGED line.
```

Adding New Text to a File

Before discussing how to change text, let's spend more time on adding new text to a file. If the file is empty (new edit on a file that does not already exist), the cursor is at the beginning of the first line and cannot be moved because there are no additional characters in the file. If you are editing an existing file, the cursor must be moved to the location in the file where the change is to be made before you select an editing command to add new text. Here are the *vi* commands that can be used to add new text to a new or existing file:

Commands for Adding Text to a File

Command	Result
i	Insert new text in front of current cursor character until [ESC] is pressed.
I	Insert new text in front of the first visible character in the current line until [ESC] is pressed.
a	Append new text after the current cursor character until [ESC] is pressed.
A	Append new text following the last character on the current line until [ESC] is pressed.
o	Open a new line after the current line and add new text until [ESC] is pressed.
O	Open a new line above the current line and add new text until [ESC] is pressed.

When any of these commands is used, there is no limit on the number of characters that can be added. You can add zero characters by pressing [ESC] immediately, add a few or several characters, or add many lines of new text. Here are some examples that demonstrate the effect of each command. Consider the following sample line of text where the underscore character represents the current cursor location:

```
This is the starting sentence.
          ^
```

In each example that follows, the text being added does not vary; only the command changes between examples. Only the characters shown are typed; there are no leading or trailing blanks (spaces) before or after the new text:

```
ADDED TEXT[ESC]
```

Using the **i** command:

```
This is theADDED TEXT starting sentence.
```

Using the **I** command:

```
ADDED TEXTThis is the starting sentence.
```

Using the **a** command:

```
This is the ADDED TEXTstarting sentence.
```

Using the **A** command:

```
This is the starting sentence.ADDED TEXT
```

Using the **o** command:

```
This is the starting sentence.
ADDED TEXT
```

Using the **O** command:

```
ADDED TEXT
This is the starting sentence.
```

ASCII Control Characters in Text

Many common situations require the ability to insert or deal with ASCII control characters as part of the normal text body. The need may arise in a computer program, or be as a simple as the insertion of a form-feed or other character being used to control devices or programs that interact with the file when it is used or processed in the future.

Control Characters Defined

ASCII is an acronym that stands for *American Standard Code for Information Interchange.* A particular pattern of binary digits (bits) establishes a code pattern that defines a corresponding character. Thus a code value equivalent to 106 decimal represents lowercase **j** and the code value equivalent to decimal 82 represents uppercase **R.** Most ASCII character codes produce visible text when printing or editing. However, certain characters such as form-feed (FF), carriage-return (CR), end-of-transmission (EOT), and such do not produce a visible printed character because their functions are related to data handling and formatting instead of forming words.

Obtaining Control Characters

Control characters are usually typed from the terminal keyboard by pressing a normal typing character key while holding the CTRL key down, thus producing a control character. However, many control characters represent *vi* editor commands, making it impossible to enter them directly into text. For example, CTRL-H and the BACK SPACE key both generate a backspace character which is used as a backspace command while in insert/append mode. If you needed to include an ASCII backspace character in the text file being edited, the editor would interpret the character as a backspace command and act accordingly instead of placing it in the text file. Other control characters are also interpreted as editor commands, either during insert/append mode or while in command mode. For example, CTRL-D and CTRL-F are used for screen manipulation in command mode.

Displaying Control Characters

When *vi* or HP-UX displays control characters, each control character is displayed as a combination of two characters: a circumflex (^) character representing the CTRL key followed by the typing key that is pressed simultaneously with the CTRL key to obtain the control character. For example, CTRL-K, which produces a vertical tab character is displayed in text on the display screen as **^K**. However, the two displayed characters only represent the single control character that exists in the file being edited. Thus when you move the cursor along a line containing a control character, you will discover that the cursor skips over the circumflex character and stops on the uppercase character that follows it, thus identifying the character as a single control character rather than two ASCII printable characters.

A complete list of control characters and the keypress combinations required to produce them is contained in Table 4-1 later in this section.

Entering Control Characters

When adding new text during insert, append, replace, substitute, and similar operations with *vi*/*ex*, press CTRL-V (think "verbatim" to remember the **V**) to tell the editor that the next character is a text character instead of an editor control command (this rule applies both in insert mode and in regular expressions). *vi* acknowledges the CTRL-V by placing a circumflex character (^) on the screen at the cursor position. When you type the next control character, the cursor advances one column and the typing character used with the CTRL key is displayed. If the next character is not a control character (CTRL key was not held down while typing key was pressed), the circumflex character from the CTRL-V is removed and the typed character is displayed in its stead. Thus you can change your mind after typing the ^V and continue typing normal text without any special procedures to abort the control character set-up.

Control characters in text are easily differentiated from a circumflex text character followed by another character. While in command mode, use the space bar or L to advance the cursor across the character(s). If the cursor stops on the circumflex when a cursor advance key is pressed, it is ordinary text. If the cursor skips the circumflex and stops on the following character, a control character is located at that position in the file.

Important

Control characters used in terminal-computer handshaking such as ENQ, ACK, etc., are consumed by the datacomm interfacing hardware and are not transmitted as part of the text. Thus they cannot be used. This also means that CTRL-F cannot be used for scrolling on an ENQ/ACK protocol connection because it is an ACK character.

Selecting Control Characters

Understanding how to enter control characters is all well and good, but one cannot conveniently use them without knowing which typing key produces a given control character. Few manuals document the relationship between a typed key and its corresponding control character (obtained when CTRL is pressed at the same time), so the next page contains a useful table for making the conversion with little effort. Columns 1, 2, and 3 in the table show octal, decimal, and hexadecimal numerical equivalents for the ASCII character code whose acronym is shown in the fifth column. Column 4 contains the character code as displayed on a terminal screen by *vi*, and Column 6 contains the full name of the character and the right-hand column lists the keys that must be pressed simultaneously to produce the control character described on that line.

For a complete list of ASCII character codes, refer to the *ascii*(5) manual entry in the *HP-UX Reference*. For European languages, see *roman8*(5) instead. Control character codes are identical in both character sets.

Note

The table of ASCII control codes on the next page lists 32 ASCII control characters and provides a way to obtain each one. However, since there are many more typing keys on a keyboard than there are control characters, most control characters can be obtained by several different key combinations. For example, CTRL-SHIFT-~ and CTRL-SHIFT-^ both produce the record separator character that is also used to switch between two files that are being edited simultaneously as described in Chapter 11.

Table 4-1: Typical Typing-Key to Control-Character Conversions

Oct	Dec	Hex	Dsp	Symbol	Character Name	Keypress
000	000	00	none	NUL	Null	CTRL-SHIFT-@
001	001	01	^A	SOH	Start of Header	CTRL-A
002	002	02	^B	STX	Start of Text	CTRL-B
003	003	03	^C	ETX	End of Text	CTRL-C
004	004	04	^D	EOT	End of Transmission	CTRL-D
005	005	05	^E	ENQ	Enquire	CTRL-E
006	006	06	^F	ACK	Acknowledge	CTRL-F
007	007	07	^G	BEL	Bell	CTRL-G
010	008	08	^H	BS	Back Space	CTRL-H
011	009	09	^I	HT	Horizontal Tab	CTRL-I
012	010	0A	^J	LF	Line Feed (newline)	CTRL-J
013	011	0B	^K	VT	Vertical Tab	CTRL-K
014	012	0C	^L	FF	Form Feed (newpage)	CTRL-L
015	013	0D	^M	CR	Carriage Return	CTRL-M
016	014	0E	^N	SO	Shift Out	CTRL-N
017	015	0F	^O	SI	Shift In	CTRL-O
020	016	10	^P	DLE	(or DEL) Delete	CTRL-P
021	017	11	^Q	DC1	Device Control 1	CTRL-Q
022	018	12	^R	DC2	Device Control 2	CTRL-R
023	019	13	^S	DC3	Device Control 3	CTRL-S
024	020	14	^T	DC4	Device Control 4	CTRL-T
025	021	15	^U	NAK	Negative Acknowledge	CTRL-U
026	022	16	^V	SYN	Synchronize	CTRL-V
027	023	17	^W	ETB	End Transmission Block	CTRL-W
030	024	18	^X	CAN	Cancel	CTRL-X
031	025	19	^Y	EM	End of Medium	CTRL-Y
032	026	1A	^Z	SUB	Substitute	CTRL-Z
033	027	1B	^[	ESC	Escape Code	CTRL-[
034	028	1C	^\	FS	File Separator	CTRL-\
035	029	1D	^]	GS	Group Separator	CTRL-]
036	030	1E	^^	RS	Record Separator	CTRL-SHIFT-^
037	031	1F	^_	US	Unit Separator	CTRL-SHIFT-_
177	127	7F	^_	DEL	Delete	CTRL-V DEL[1]

[1] To obtain the DEL character, press CTRL-V, then press DEL.

CAUTION

Certain control characters cannot be used in creating and editing text because they are used in datacomm protocol and terminal control functions. Which characters fall into this control category and become unusable as a result depends on the brand and model of terminal being used as well as datacomm line protocol. Refer to the terminal operating manual for more information.

8-Bit Control Characters

When editing files containing 8-bit characters, the following table shows how 8-bit control characters are displayed on the terminal display screen. These control characters cannot be entered directly from the terminal keyboard, but *vi* does process them correctly if they exist in the file.

Table 4-1: Display Representation for 8-bit Control Characters

Oct	Dec	Hex	Dsp	Oct	Dec	Hex	Dsp
177	127	7F	^?	220	144	90	^p
200	128	80	^‘	221	145	91	^q
201	129	81	^a	222	146	92	^r
202	130	82	^b	223	147	93	^s
203	131	83	^c	224	148	94	^t
204	132	84	^d	225	149	95	^u
205	133	85	^e	226	150	96	^v
206	134	86	^f	227	151	97	^w
207	135	87	^g	230	152	98	^x
210	136	88	^h	231	153	99	^y
211	137	89	^i	232	154	9A	^z
212	138	8A	^j	233	155	9B	^{
213	139	8B	^k	234	156	9C	^|
214	140	8C	^l	235	157	9D	^}
215	141	8D	^m	236	158	9E	^~
216	142	8E	^n	237	159	9F	^>
217	143	8F	^o	377	255	FF	^/

Changing Text: Overview

Most text change operations involve fairly simple operations such as correcting a typographical error or changing the wording in a sentence or restructuring a line of computer program source code. *vi* provides a useful set of tools for performing various combinations of common and not-so-common text alterations. Here are a few topics that are discussed in this chapter:

- Delete text. Deleted text is copied into a buffer so that it can be used elsewhere in the file.
- Replace existing text with new text by using **replace**, **substitute**, or **change** commands.
- Copy (**yank**) existing text into a buffer so that it can be used elsewhere in the file.
- Copy text from a yank or delete buffer into the current cursor location by using the **put** command. A **delete** and **put** sequence is used to move blocks of text to a different location; a **yank and put** sequence copies text from one location to another. Under certain conditions, these commands can be used to copy or move text from one file to another as described in the advanced topics chapters of this manual.

Command Format

Like the HP-UX and other similar operating systems in general, *vi* usually provides several ways for performing any given task. This fact will become readily apparent as you learn more about the editor and its many capabilities. This section shows several commands that are, for most practical purposes, essentially identical or very similar. A significant effort has been invested in making those similarities visible for ease in learning.

All *vi* commands related to deleting, changing, copying, or moving text objects have a form that, if understood beforehand, makes them much easier to learn and use. Each command is a variation on the following two structures:

⟨count1⟩ **command**

or

⟨count1⟩ **command** ⟨count2⟩ *text_object*

These two simple structures support a vast selection of editing options which are described in greater detail in the remainder of this chapter.

Character and Line Oriented Commands

Commands that have the form:

⟨count1⟩ **command**

are generally commands related to character- or line-oriented operations such as:

- Deleting one or more characters or lines by specifying the number of characters or lines to delete,
- Replacing one or more characters or lines with new text by specifying the number of characters or lines to replace, or
- Moving or copying text to another location (however, certain common text copy/move operations require the text object form described next).

Word, Sentence, and Text Object Oriented Commands

On the other hand, commands that use the form:

⟨count1⟩ **command** ⟨count2⟩ *text_object*

are generally commands related to specified text objects. Text objects are words, sentences, paragraphs, sections, or all text between two marked or known locations in the file. Editing operations performed on text objects include:

- Deleting one or more text objects such as words, sentences, paragraphs, or all text from current position to another specified position.
- Replacing one or more text objects with new text by specifying the type and number of text objects to replace,
- Moving or copying one or more text objects to another location by use of buffers.

Note the similarities between this list and the one preceding.

How Text Objects are Defined

In general, a text object is defined as all text between the current cursor position and another position in the file that is specifed by a cursor move command from the current cursor location. Thus, if a delete command is followed by a command to move the cursor forward nine words, the editor deletes nine words starting with the current cursor character which may or may not be at the beginning of the current word.

Text Object Boundaries

In general, text objects such as words, sentences, paragraphs, and sections are bounded at the start of an object where the starting point is the boundary between the first character in the object and the preceding character. Thus, in the text string **one two three**, the word **two** begins at the boundary between the **t** and the space preceding it.

Likewise, when the current cursor position is used as a text object boundary, the object boundary is at the boundary between the cursor character and the character preceding it. This means that if a change, deletion, or yank is from the current cursor position in the forward direction, the current cursor character is included in the change because it falls within the object as the first character in the object. On the other hand, if the change, deletion, or yank is in the reverse direction toward beginning of file, the cursor character falls outside the boundary and is not included in the change because the text object starts in the boundary region before the cursor character and progresses away from it toward beginning of file.

The Commands and What They Do

This section explains the behavior of individual text manipulation commands based on the type of operation. Detailed examples of how to use many of these commands follow later in the chapter.

Deleting Characters and Lines

When deleting text, the editor performs the deletion as soon as it has sufficient information to determine the operation to be made. Pressing ESC is not necessary.

Delete Character(s)

Command	Action Taken
x	Delete single character at current cursor position.
*n***x**	Delete *n* characters or to end-of-line, whichever occurs first, starting at current cursor position
X	Delete single character immediately preceding current cursor position.
*n***X**	Delete *n* characters or to beginning-of-line, whichever occurs first, starting with the character immediately preceding the current cursor position.
D or **d$**	Delete all characters from current cursor position to end of line.
d0 or **d|**	Delete all characters from left column of screen to character preceding current cursor position on current line.

Delete Line(s)

Command	Action Taken
dd	Delete current line.
*n***dd** or **d***n***d**	Delete *n* lines beginning at current line.
dG	Delete all lines, starting with current line, through end-of-file.
d1G	Delete all lines, starting with current line, through beginning-of-file.
d*n***G**	Delete all lines, starting with current line, through line *n* in file (forward or backward, depending on position of line *n* relative to current line).
d−	Delete current and first preceding line.[1]
d+	Delete current and first following line.[1]
*n***d−** or **d***n***−**	Delete current and *n* previous lines.[1]
*n***d+** or **d***n***+**	Delete current and *n* following lines.[1]

[1] In these commands, **k** can be used instead of **−**, and **j** or RETURN can be used instead of **+**.

Deleting Text Objects

Delete Word or Part of Word

Command	Action Taken
dw	Delete from cursor position through end of current word[1].
dW	Delete from cursor position through end of current Word[1].
d*n***w** or ***n*dw**	Delete from cursor position through *n*th following word[1] ending.
d*n***W** or ***n*dW**	Delete from cursor position through *n*th following Word[1] ending.
db	Delete from nearest preceding beginning[1] of word through character before current cursor position.
dB	Delete from nearest preceding Beginning[1] of word through character before current cursor position.
d*n***b** or ***n*db**	Delete from *n*th preceding beginning[1] of word through character before current cursor position.
d*n***B** or ***n*dB**	Delete from *n*th preceding Beginning[1] of word through character before current cursor position.

Dealing with Whitespace

When deleting words, any whitespace between the word being deleted (or the last word if multiple words are being deleted) and the word following is also deleted. If the last word being deleted is at the end of the line, the end of line remains after the word preceding the deleted text. If one or more words beyond the end of the current line are being deleted, the following line is appended to the current line during the deletion. If the word count in a multiple-word deletion spans more than two lines, additional lines are appended to the current line prior to the deletion until the deleted word count specification is satisfied. For deletions toward beginning of file, the beginning of the removed text block is calculated, then the deletion proceeds from that point toward end of file.

[1] When lowercase **b** or **w** is used, any character other than alphanumeric or underscore is considered as the beginning of a new word except whitespace characters that are treated as word separators.

When uppercase **B** or **W** is used, word boundaries are defined by whitespace word separators. Any other non-alphanumeric characters (including underscore and control characters) are treated as part of the word.

Note

Deleting text objects such as words, sentences, paragraphs, and such are not restricted to the current line. If the number of text objects specified exceeds current line contents, the object is extended until the text specification is completely satisfied.

Note

Word boundaries are defined based on beginning of word. Thus in deleting a word in the forward direction **dw** or **dW**, if any whitespace exists between the current position and the start of the next word, it is deleted. If the deletion is toward beginning of file (**db** or ⁻dB), only whitespace between the current position (not including current character) and the target beginning of word is deleted.

Delete All or Part of Sentence, Paragraph, or Section

Command	Action Taken
d)	Delete from cursor position through first following end of sentence.
d}	Delete from cursor position through first following end of paragraph.
d]	Delete from cursor position through first following end of section.
d*n***)** or ***n*d)**	Delete from cursor position through *n*th following end of sentence.
d*n***}** or ***n*d}**	Delete from cursor position through *n*th following end of paragraph.
d*n***]** or ***n*d]**	Delete from cursor position through *n*th following end of section.
d(	Delete from closest previous start of sentence through character before cursor.
d{	Delete from closest previous start of paragraph through character before cursor.
d[	Delete from closest previous start of section through character before cursor.
d*n***(** or ***n*d(**	Delete from *n*th preceding start of sentence through character before cursor.
d*n***{** or ***n*d{**	Delete from *n*th preceding start of paragraph through character before cursor.
d*n***[** or ***n*d[**	Delete from *n*th preceding start of section through character before cursor.

Deleting to a Text Location in Line or File

Text is deleted from current cursor position to a specified character on current line or a specified text pattern in the file being edited. The differences between **f** and **t** and between **F** and **T** are subtle, but useful in certain situations.

Delete Through Character on Current Line

Command	Action Taken
df*c*	Delete text from current position through first occurrence of character *c* on the current line when scanning toward end of line.
d*n***f***c*	Delete text from current position through *n*th occurrence of character *c* on the current line when scanning toward end of line.
dF*c*	Delete text from first occurrence of character *c* on the current line when scanning toward beginning of line to character preceding cursor.
d*n***F***c*	Delete text from *n*th occurrence of character *c* on the current line when scanning toward beginning of line to character preceding cursor.

Delete Up To a Given Character on the Current Line

Command	Action Taken
dt*c*	Delete text from current position to first occurrence of character *c* on the current line when scanning toward end of line.
d*n***t***c*	Delete text from current position to *n*th occurrence of character *c* on the current line when scanning toward end of line.
dT*c*	Delete text from character following first occurrence of character *c* on the current line when scanning toward beginning of line to character preceding cursor.
d*n***T***c*	Delete text from character following *n*th occurrence of character *c* on the current line when scanning toward beginning of line to character preceding cursor.

Delete Text from Current Position to a Specifed Text Pattern

In the following descriptions, *search_pattern* is any valid HP-UX regular expression recognized by standard HP-UX editors and other commands. Refer to the tutorial on Regular Expressions earlier in this volume for more information.

Command	Action Taken
d/*search_pattern* RETURN	Delete all text from current location to first occurrence of text matching *search_pattern* when searching in forward direction toward end of file. If *search_pattern* is matched before end of file is reached, deletion is from current cursor character up to but not including the matched text pattern. If search wraps to beginning of file before the pattern is matched, deletion begins with text pattern and all text is removed up to, but not including, the current cursor character.
d?*search_pattern* RETURN	Delete all text from current location to first occurrence of text matching *search_pattern* when searching in reverse direction toward beginning of file. If *search_pattern* is matched before beginning of file is reached, deletion is from start of text that matches *search-pattern* up to, but not including, current cursor character. If search wraps to end of file before the pattern is matched, deletion begins with the current cursor character and continues up to, but not including, the matching text pattern.

Note

In general, the current cursor character is included in all changes and deletions in the forward direction. If the change or deletion is in the reverse direction, the current cursor character remains undisturbed.

Text Delete/Change Command Examples

The following examples show various forms of text **delete** commands. They apply equally to **c**hange or **y**ank commands by replacing the **d** in each command with **c** or **y** respectively.

- **4dd**, **d4d**, **3d+**, and **d3+** are equivalent. Each removes 4 lines starting with current line.
- **d5w** deletes five words starting at current cursor position. If there is only one word left on current line, the remaining four words are deleted on the following line or lines until the count of five words is filled. Counts related to sentences, paragraphs, and sections are handled the same way.
- **word1,word2,word3** is one **W**ord or five **w**ords (each comma is treated as a separate word). **This_is_one_word** is treated as a single **w**ord.
- If cursor is in mid-sentence, **d)** deletes rest of sentence. To delete entire sentence use **(d)** where **(** moves cursor to beginning of sentence. Same technique applies for paragraphs and sections.
- If cursor is at beginning of sentence, **d)** deletes entire sentence. On the other hand, **(d)** deletes the previous sentence because **(** moves cursor to next previous beginning-of-sentence and the **d)** deletes to the next following end-of-sentence. Same technique applies for paragraphs and sections.
- **(d)** and **)d(** are functionally equivalent if cursor is not at beginning of sentence. One moves cursor to beginning of sentence and removes to following end of sentence; the other moves cursor to end of sentence and removes text starting at previous beginning of sentence. The same logic applies when deleting or changing paragraphs or sections.

Recovering Deleted Text

Whenever a change, delete, or yank command is executed on a text object, the object is copied into a buffer where it can be easily recovered. The two commands used to recover text from the buffer are the "put" commands, **p** and **P**:

Command	Action Taken
p	Put buffer contents in text **after** current cursor position.
P	Put buffer contents in text **before** current cursor position.

The buffer may contain part of a line or it may contain one or more lines. Consequently, the transfer of buffer text must be handled according to the nature of the text being copied from the buffer. Generally speaking, text is copied from the buffer as follows:

- If the original command that placed text in the buffer was a change, delete, or yank lines command, buffer text is copied into the file being edited immediately before or immediately after the current cursor line as dictated by the **p** (after current line) or **P** (before current line) command.
- For most other changes, deletions, or yanks (characters, words, sentences, paragraphs, sections, etc.), buffer text is copied into the current line in front of or immediately after the current cursor character as dictated by the **p** (after current character) or **P** (before current character) command.

It is this technique that gives rise to the **xp** command for swapping the positions of the current and next following character on the current line or the **ddp** command for swapping the position of the current and following line in a file.

Text can be easily copied or moved from one location to another by using a delete or yank command to place text in a buffer, moving the cursor to a new location in the file, then using **p** or **P** to place the deleted or yanked text in the new location. The **p** and **P** commands only copy the buffer into text, so the buffer contents is not disturbed. This means you can easily move to another location in the file and use another put command to repeat the operation as many times as you choose.

This method is described in greater detail in Chapter 6 which discusses moving and copying text in a file.

Using Named Buffers for Deleted or Yanked Text

In addition to the default buffer, *vi* provides 26 named buffers that can be used for copying and moving text objects. Again, these buffers and how they are used in conjunction with delete and yank commands are discussed in detail in Chapter 6 which deals with copying and moving blocks of text.

Changing Text

When the editor is given a text change command, it enters replace or insert mode as soon as it can determine the type of operation being performed, then accepts new input text until [ESC] is pressed. The only exception is the **r** or *n***r** command form which accepts only one character of replacement text and does not require an [ESC] character to terminate the operation.

Replace Text Character(s)

Command	Action Taken
~	Change cursor character from lowercase to uppercase or vice-versa.
n~	Identical to ~ command (*n* is ignored).
r	Replace single character at current cursor position (no [ESC] needed).
*n***r**	Replace single character at current cursor position with *n* copies of replacement character (no [ESC] needed).
R[1]	Replace text, character by character, starting at current cursor position.
*n***R**[1]	Change *n* characters or to end-of-line, whichever occurs first, starting with character immediately preceding current cursor position.
s	Replace single character at current cursor position with new text.
*n***s**	Replace *n* characters (or to end of line if it occurs before *n* characters) starting at current cursor position with new text.
C or **c$**	Change all characters from current cursor position to end of line.
c0 or **c|**	Change all characters from left column of screen to character preceding current cursor position on current line.

[1] When overstriking characters in conjunction with the **R** or *n***R** command, if you encounter a tab character, the tab character is not replaced by a single replacement character, but rather, the tab is expanded on the display screen and the number of characters that replace the tab is determined by the number of spaces used to expand the tab character for display purposes.

To abort a character replacement following an **r**, *n***r**, **R**, or *n***R** command, press [ESC] instead of the replacement character. To abort a substitute or change command, press [ESC], then lowercase [u] to undo the text removal that results from the [ESC].

Change Line(s)

Command	Action Taken
cc	Change current line.
*n***cc** or **c***n***c**	Change *n* lines beginning at current line.
cG	Change all lines, starting with current line, through end-of-file.
c1G	Change all lines, starting with current line, through beginning-of-file.
c*n***G**	Change all lines, starting with current line, through line *n* in file (forward or backward, depending on position of line *n* relative to current line).
c−	Change current and preceding line.
c+	Change current and following line.
*n***c−** or **c***n***−**	Change current and *n* previous lines.
*n***c+** or **c***n***+**	Change current and *n* following lines.

To abort a change command, press [ESC], then lowercase [u] to undo the text removal that results from the [ESC].

Change Text Objects

Change Word or Part of Word

Command	Action Taken
cw	Change from cursor position through end of current **word**[1].
cW	Change from cursor position through end of current **Word**[1].
c*n***w** or ***n*cw**	Change from cursor position through *n*th following **word**[1] ending.
c*n***W** or ***n*cW**	Change from cursor position through *n*th following **Word**[1] ending.
cb	Change from nearest preceding **beginning**[1] of word through character before current cursor position.
cB	Change from nearest preceding **Beginning**[1] of word through character before current cursor position.
c*n***b** or ***n*cb**	Change from *n*th preceding **beginning**[1] of word through character before current cursor position.
c*n***B** or ***n*cB**	Change from *n*th preceding **Beginning**[1] of word through character before current cursor position.

Note

When changing words, *vi* assumes that whitespace should not be disturbed. Consequently, the change is from the boundary between the current cursor character and the character preceding it and the destination defined by the number and type of object(s) to be changed (word, sentence, paragraph, etc.). Any whitespace preceding or following the object to be changed remains unaltered.

[1] When lowercase **b** or **w** is used, any character other than alphanumeric or underscore is considered as the beginning of a new word except whitespace characters that are treated as word separators.

When uppercase **B** or **W** is used, word boundaries are defined by whitespace word separators. Any other non-alphanumeric characters (including underscore and control characters) are treated as part of the word.

Change All or Part of Sentence, Paragraph, or Section

Command	Action Taken
c)	Change from cursor position through next end of sentence.
c}	Change from cursor position through next end of paragraph.
c]]	Change from cursor position through next end of section.
c(	Change from preceding start of sentence through character before cursor.
c{	Change from preceding start of paragraph through character before cursor.
c[[	Change from preceding start of section through character before cursor.

Note

Changing text objects such as words, sentences, paragraphs, and such are not restricted to the current line. If the number of text objects specified exceeds current line contents, the object is extended until the text specification is completely satisfied.

To abort a change command, press [ESC], then lowercase [u] to undo the text removal that results from the [ESC].

Change Text between Two Boundaries in Line or File

Text from current cursor position to a specified character on current line or a specified text pattern in the file being edited is replaced with new text until [ESC] is pressed.

Change Text Up To and Including a Given Character on Current Line

Command	Action Taken
c*f*c	Change text from current position through first occurrence of character *c* on the current line when scanning toward end of line.
c*n*f*c*	Change text from current position through *n*th occurrence of character *c* on the current line when scanning toward end of line.
c**F***c*	Change text from current position through first occurrence of character *c* on the current line when scanning toward beginning of line.
c*n*F*c*	Change text from current position through *n*th occurrence of character *c* on the current line when scanning toward beginning of line.

Change Text Up To but not Including a Given Character on Current Line

Command	Action Taken
c**t***c*	Change text from before current position up to first occurrence of character *c* on the current line when scanning toward end of line.
c*n*t*c*	Change text from before current position up to *n*th occurrence of character *c* on the current line when scanning toward end of line.
c**T***c*	Change text from before current position up to first occurrence of character *c* on the current line when scanning toward beginning of line.
c*n*T*c*	Change text from before current position up to *n*th occurrence of character *c* on the current line when scanning toward beginning of line.

Change Text from Current Position to a Specifed Text Pattern

Command	Action Taken
c/*search_pattern* RETURN	Change all text from current location to first occurrence of text matching *search_pattern* when searching in forward direction toward end of file. If *search_pattern* is matched before end of file is reached, text change is from current cursor character up to but not including the matched text pattern. If search wraps to beginning of file before the pattern is matched, change begins with text pattern and all text is removed up to, but not including, the current cursor character.
c?*search_pattern* RETURN	Change all text from current location to first occurrence of text matching *search_pattern* when searching in reverse direction toward beginning of file. If *search_pattern* is matched before beginning of file is reached, change is from start of text that matches *search-pattern* up to, but not including, current cursor character. If search wraps to end of file before the pattern is matched, change begins with the current cursor character and continues up to, but not including, the matching text pattern.

Repeating a Text Change Operation

vi provides a "dot" command (.) that tells the editor to repeat the last operation that resulted in a text change. It can be used after a delete, replace, change, yank/put, or any other command that changes text.

It is most commonly used when making a series of identical or very similar changes throughout a file without typing the text more than once. It is especially useful when using the search commands / and ? together with the repeat-search commands **n** and **N**. To repeat a search for a given text pattern in a file, use **n** (repeat search in same direction) or **N** (repeat search in opposite direction). Adjust the cursor position if necessary, then press . (period or "dot") to repeat the last change.

For example, suppose you are building a list of file names preceded by a common pathname for use in a file manipulation script or a system document. Here is a sample of fictitious text:

```
This program block contains the following files:

Be sure all files are present before compiling the package.
```

You now need to add file and pathnames to the text from a handwritten list. Placing the cursor anywhere in the line preceding the blank line, type the **o** command followed by the path and file name: */users/prog_mgr/systemA/fileset1* then press ESC. The result becomes:

```
This program block contains the following files:
/users/prog_mgr/systemA/fileset1/file1

Be sure all files are present before compiling the package.
```

By pressing . three times after ESC is pressed, the result looks like this:

```
This program block contains the following files:
/users/prog_mgr/systemA/fileset1/file1
/users/prog_mgr/systemA/fileset1/file1
/users/prog_mgr/systemA/fileset1/file1
/users/prog_mgr/systemA/fileset1/file1

Be sure all files are present before compiling the package.
```

Now it is a simple matter to move the cursor to the last word on each of the second, third, and fourth lines and edit the filename to obtain the other desired names. You can also get three more copies of the yanked line by typing **yy** followed by **p..** or **ppp** after pressing ESC.

With a little experimentation, you can quickly become proficient in the use of the frequently used command.

Using Numbered Buffers to Restore Text

vi maintains a delete/change buffer that can be used to restore the last change by means of the put (**p** or **P**) command. However, the put command, by itself is only able to restore the most recent changed, yanked, or deleted text.

vi also maintains a history of changes/deletions in a group of numbered buffers, 1 through 9. They contain the preceding deletions and/or changes in a last-in, first-out push-down stack arrangement. This means that buffer 1 contains the most recent text while buffer 9 contains the least recent. To show how they are used, consider the following text:

```
line 1
line 2
line 3
line 4
line 5
line 6
line 7
line 8
line 9
```

Restoring Changes/Deletions in Reverse Order

Place the cursor on the first line, **line 1**, and type the **dd** command to delete the line. Now, press **.** eight times in succession to delete the remaining lines in the series. To restore the most recent deletion, type a double quote (**"**) followed by the buffer number (**1**), and the lowercase **p** command (none of the typed characters appear on the display). Notice how the last line deleted now appears on the next line after the cursor and the cursor is moved to that line. Now, press **.** eight times to get this result:

```
line 9
line 8
line 7
line 6
line 5
line 4
line 3
line 2
line 1
```

This technique shows that the buffer pointer is advanced to the next buffer each time a put command is executed provided the buffer number is used for the first put and **.** is used for successive operations. The recovered line is placed **after** the current line, thus producing the reversed order.

But I Don't Want Them in Reverse Order

You can also do the same without reversing the order. After making the last deletion, use the command **"1P** (note the uppercase **P**) followed by **.** eight times to get this:

```
line 1
line 2
line 3
line 4
line 5
line 6
line 7
line 8
line 9
```

In this example, recovered lines are placed before the current line such that the lines are restored in their original order.

You can also specify any buffer as you need it. For example, **"6p** restores the buffer containing the text **line 4** after the current line if you use the previous deletion example. Specifying any number greater than 9 restores the most recent deletion.

Using the Commands

The remainder of this chapter shows various examples that illustrate how to use many of the commands previously discussed for deleting and altering text. In many cases, several approaches to a given problem are used to demonstrate the flexibility of a large editor program like *vi*.

Examples of Deleting and Swapping Characters

Vi's flexibility usually offers several ways to accomplish a given task. The approach you use will usually depend a great deal on how familiar you are with various methods and techniques as well as your particular interests and preference. For example, consider the following line of text where errors have been marked by a circumflex on the next line below:

```
For exammple, Consider the follollowing line of txeT.
        ^                      ^^^                  ^^
```

We could tell you that the easiest way to fix the line is to simply move the cursor to near the beginning of the line then type the following characters:

fmxfl3xfxxp

but you probably would wonder what all those characters mean, so let's take some time and learn how to attack a problem of this nature and determine what methods work best.

Four words need to be changed. A fast typist could move the cursor to the first error, clear the rest of the line, and retype it by using a **D** command followed by an **a** or **A** command. A less proficient user would probably prefer other methods requiring fewer keystrokes. Let's look at our options.

Deleting Characters

Assuming that the cursor is located at the beginning of the line and *vi* is in Command Mode, use the space bar (or the **l** key) to move the cursor to either **m** in **exammple.** Deleting a single character corrects the misspelling, so press **x** to remove the character at the current cursor location (uppercase **X** removes the character preceding the cursor character).

Now let's try a second, faster method. Press **u** to undo the change, then press **0** (zero) to move the cursor to the beginning of the line. Now, type **fm**. What happened? Why? Press **0** again. Now type **2fm**. What is different? Why? Press **x** to remove the current cursor character.

The sentence should now look like this:

```
For example, Consider the follollowing line of txeT.
```

Now let's work on that gruesome word `follollowing`. Before we can fix it, the cursor has to be moved to the characters that should be removed (the first or second `llo`). First, an explanation of several ways to do it:

1. We have already learned that using **l** or the space bar is not always the quickest way to move through a line. It is adequate for up to a half-dozen or so characters, but for longer moves it is usually too slow.

2. The word-skip command is available to move over words much like **l** moves over characters. Here is how it can be used:

   While observing the cursor, type **w**. The cursor moves to the comma because it is a non-alphanumeric character that is treated as the beginning of the next word. Press **w** again. The cursor moves to the beginning of `Consider` (the comma and space are treated as a single word). Now type **Fp** to move the cursor back to the previously deleted character position. Press **W**. Notice how the cursor skips over the comma this time and moves to the beginning of the next full word. **W** tells *vi* to use only white space to detect the end of a word, and include non-alphanumeric characters as part of the word being skipped over (use of **w** and other similar cursor control commands are discussed in detail in the advanced topics section of this tutorial).

   Return the cursor to the previously deleted character (**Fp**), then type **3W**. Note how the cursor moves over three full words (ignoring punctuation), and stops at the beginning of `follollowing`. Now move the cursor back where it was and type **4w**. Notice that you get the same result as using **3W**. Use the space bar or **l** to move the cursor to the first `llo` in `follollowing`, then type **3x** to delete three characters.

3. The easy way, given the characters in the sentence is to use the find command, **fl**. Note the new cursor position. Now type **FF**. Note that this gives the same result in this case as using **0**. Type **fl** again. Press **0** to return to the left column. Now type **2 fl** then **3 x**. Here is another way. Press **u** then **0** to undo the change and return to the left column, then **fl** to return to the first **l**. Now type **d f o** to delete characters from the cursor position through the first occurrence of **o**. In this situation, **d f o** is equivalent to **d 2 t l** (delete up to but not including the second occurrence of the character **l**). Simple enough?

4. Repeat step 3, but instead of using **3 x**, type **d f o**. This time it removes all characters up through the following **o** to obtain the same result.

The sentence now reads:

```
For example, Consider the following line of txeT.
```

Swapping Characters

Now let us correct the last word in the line. Press [$] to move the cursor to the last character position, then use **h** or [BACK SPACE] to move back to the **x** in **txet** (**fx** would get you there faster). Type **x** to delete the cursor character, then type **p** to add it after the **e**. Thus, to reverse two characters in text, type **xp** while in Command Mode.

Changing Uppercase/Lowercase

Now press the space bar or **l** to move the cursor to the last letter in **texT**. Press ~ (tilde) to change the uppercase **T** to lowercase. Notice that the cursor advances to the period after the reversal is completed. Press [BACK SPACE] ~. What happens? Why? Press [BACK SPACE] ~ again to restore lowercase.

The sentence now reads:

```
For example, Consider the following line of text.
```

Searching within a Line: f and F versus t and T

The cursor is now located at the end of the sentence underneath the period. We know from experience that **FC** would move the cursor to the uppercase C at the beginning of the third word in the current line. Type **TC** instead. What happened? Why? Type [$] then **FC**. Type ~ to drop C to lowercase. Now, type **tx**. Where did the cursor go? How is this different from **fx**?

The preceding pages have covered many concepts, but you now have the foundation for many skills that can be used with *vi*. Take some time to practice them on a few sentences of your own. The time will be amply rewarded as you move toward editing usable text.

Using Semicolon and Comma to Repeat a Search Within a Line

As you become more familiar with using **f** and **F** to search for characters in a line, you will discover that you frequently execute a search for a character (such as **5fl**) only to find that you miscounted and got the wrong occurrence. Rather than typing another search command, it is usually easier to press semicolon (;) as a command to find the next occurrence of the character, possibly pressing it twice or more to get the correct one.

If you overshoot the character you want in a forward search, you can easily back up to the previous occurrence by using the comma (,) command. Thus **5fw ,** is equivalent to **4fw**.

Both the comma and semicolon can be preceded by a number (such as **3;** or **2,**) to repeat the search more than once. Thus, **7fw 3,** is equivalent to **4fw**, and **3fw 3;** is equivalent to **6fw**.

The semicolon or comma repeat search command always works in any form with **f** and **F**. However, if used with **t** or **T**, the semicolon must be preceded by a number larger than 1 to obtain meaningful movement. If only the next occurrence of the search character is specified by the absence of a number greater than 1, the cursor does not move after the semicolon command because the search looks for the next occurrence of the character specified in the search, then moves to the previous or following character (depending on whether the search is forward or backwards). When the search is repeated by using **;**, the character found in the previous search is again encountered, but when the cursor placement is determined, the result places the cursor in its location prior to the **;** command, resulting in no movement. On the other hand, if the repeat command is preceded by a number larger than one, the search is made for the *n*th occurrence of the search character, then the cursor returns to its correct position next to the specified character.

Examples of Replacing Text in a Line

We continue our discussion of editing within a line by using another example sentence with several errors:

```
Dis is a vewy junquey excyuse for a sentense, but wi'll uze it anyweigh.
```

Replacing a Single Character with Multiple Characters

This time, let's make the corrections in order, left to right. The first error is in the first word, so position the cursor at the first character in the line. We could use **x** to delete the first character, then insert two more characters to make the correction, but it is easier to use the **s** command. With the cursor positioned at the the **D** in **Dis**, press **s**. *Vi* places a dollar sign (**$**) at the cursor location, indicating that this character will be replaced with zero or more characters (until you press ESC to restore Command Mode). If you press ESC without entering any characters, the character space is deleted (same as the **x** command earlier). To correct the error, type **Th** and press ESC. Notice how the remainder of the line was pushed to the right to accommodate new characters after the first was replaced.

Replacing Multiple Characters with Zero or More New Characters

The **s** command can be preceded by a numeric value to specify the number of characters that are to be changed. Thus, **10s** replaces 10 characters beginning at the current cursor location with new text until [ESC] is pressed. If there are fewer than 10 characters between the cursor location and the end of the line, only the remaining characters in the line are replaced. The end-of-line position is extended, if necessary, to accommodate new characters being typed, and can continue into multiple lines if enough characters of the right kind are typed. This technique is demonstrated later in this example exercise.

Replacing a Single Character with Another

Now type **3w** to move to the next incorrect word, then press the space bar twice to move to the incorrect character and type **rr**. The first **r** tells *vi* to replace the current cursor character with the next character typed. The second **r** is the replacement character. Since only one character is being replaced, *vi* does not require an [ESC] to return to Command Mode. However, if you press **r**, then decide not to make the change, you can use [ESC] to abort the change (unlike **s**, the character is not deleted when you press [ESC]).

Our sentence now looks like this:

```
This is a very junquey excyuse for a sentense, but wi'll uze it anyweigh.
```

Replacing Multiple Characters with a Single Character

Now press **f** (for "find character") **q**. This tells *vi* to move the cursor to the next occurrence of the character **q** in the line (if the character is not present in the current line, *vi* beeps). Now type **3s**. Note that the cursor remains in its current position, but the dollar sign is located in the second column to the right. This means that new characters will be placed in the three character positions indicated until [ESC] is pressed. Type **k** and press [ESC]. Now, type **2fy**, then type **x** [ESC]. What happened and why?

Note the present cursor position in the word **excuse**. Now type slowly and watch what happens when you type **3fsrc**. What happened and why? What happens when you press [ESC]? Why? If you want to try it again, press **u** to undo the change, then move the cursor back to where it was. Our example sentence now looks like this:

```
This is a very junky excuse for a sentence, but wi'll uze it anyweigh.
```

Type **4w** [BACK SPACE] **se** [ESC]. Now type **/** (look at the bottom line of the screen) **z** [RETURN]. This operation is explained in detail later, but you just performed a forward search in the file for the next occurrence of the string expression "z". Notice how much faster and easier it would have been to simply use **fz** to accomplish the same thing. Now type **rs** to make the change.

The sentence now looks like this:

```
This is a very junky excuse for a sentence, but we'll use it anyweigh.
```

Now type **10** then press the space bar. The cursor should be located at the **e** in **eigh**. Count the number of character positions from the previous cursor position to its present position (10 total). We need to change **eigh** to **ay**. There are several ways to accomplish this, and here are a few. Try and retry all of them until you understand the principles behind each technique.

First, type **4say** [ESC]. What happened? Four characters were replaced by two. Now press **u** to undo the change and put the cursor back where it was.

Next, type **cfhay** [ESC]. What happened this time? **c** means change all characters from the current cursor position through the character position identified by the next command sequence which must follow immediately. **fh** means find the the next occurrence of the character "h", and use **ay** as replacement characters as before. [ESC] again terminates the substitution and returns *vi* to Command Mode. Press **u** again to undo the change.

Here is another way to do the same thing: Type **cway** [ESC]. As in the preceding method, **c** means change all characters from the current cursor position through the end of the "word" (as defined by our previous discussion of **w** versus **W**). Why did we use **w** instead of **W**? What happens if **W** is used instead?

The example sentence now looks like this:

```
This is a very junky excuse for a sentence, but we'll use it anyway.
```

There are many other ways to do the same thing, but these provide a useful sampling. The next topic discusses the change (**c**) command in more detail.

Changing Words Within a Line

It is sometimes necessary to reword part of a line by changing one or more words in succession. It can be tedious to use the *n***s** command because of the need to count characters before you know what *n* should be. *Vi* provides an easy solution: the change (**c**) and delete (**d**) commands which come in many varieties that were listed earlier in this chapter.

More power comes to your fingertips when certain commands are used in convenient combinations. It is much easier to swap two characters, words or lines with two to four keystrokes than to laboriously retype them.

Delete or Swap Word Commands

Command	Action Taken
dw	Delete the current word consisting of all characters from the current cursor position up to, but not including, the next non-alphanumeric character unless it is a blank or tab character (if the first non-alphanumeric character is blank or tab, it and any additional contiguous white-space excluding end-of-line is also removed).
d *n* **w**	Delete *n* words starting with the current word as defined by the **dw** command description. Count each non-alphanumeric character except blank (space), tab, and end-of-line as a separate word. Wrap to next line if necessary to match *n* word count. If the *n*th word is followed by a blank or tab, the blank or tab is also deleted, as are multiple contiguous blank/tab characters if present.
dW	Delete the word consisting of all characters from the current cursor position up to and including any following white-space characters (one or more blanks or tabs, but not end-of-line).
d *n* **W**	Delete *n* words consisting of all characters from the current cursor position up to and including, the *n*th white-space character or group of characters (blank, tab, or end-of-line or any contiguous multiple-character combination of the three). Wrap to next line if necessary to match *n* word count.
dwwP	Swap word beginning at cursor with word that follows (see **dw**). Treat non-alphanumeric characters as separate words.
dWWP	Swap word beginning at cursor with word that follows (see **dW**). Treat non-alphanumeric characters as part of the word(s) being swapped.

Let's modify our previous example sentence using some of these commands. Here is the sentence after our last changes:

```
This is a very junky excuse for a sentence, but we'll use it anyway.
```

Place the cursor at the beginning of the sentence, then type **f a** to move it to the third word **a** (you could also use **2w** if the cursor is at the beginning of first word or **3w** if the cursor was in the left column of the display screen but the sentence was indented one or more spaces – try it yourself and see).

Now, let us reword part of the sentence. Type the command **c6w**, then type:

```
not a written well[ESC]
```

To produce a reworded sentence:

```
This is not a written well sentence, but we'll use it anyway.
```

Type the command **2b** to back up two words. Now type the command **dwwP** to reverse the two words:

The sentence now reads:

```
This is not a well written sentence, but we'll use it anyway.
```

Now type (slowly) the command **b** [BACK SPACE] **r−**. What happened and why? (The **b** command is the opposite of **w** in that it moves backwards. In like manner, **B** is the counterpart of **W**.) The result is now as follows:

```
This is not a well-written sentence, but we'll use it anyway.
```

While our sentence does not reflect the literary genius of a Milton or Shakespeare, you now have a good sampling of several simple commands that are quite useful when mastered.

Changing Multiple Lines of Text

The change and delete commands can also be used with line-oriented cursor movement commands. For example, **c 10** [RETURN] replaces the complete current cursor line and the 10 following lines with new text until you press [ESC]. In like manner, **c −** replaces the current cursor line and the preceding line with new text until you press [ESC]. In the first case, **c 10** [RETURN], **c 10 +**, and **11 cc** are equivalent and can be used interchangeably.

Likewise, **d 10** [RETURN] deletes the current cursor line and the 10 following lines while **d −** deletes the current cursor line and the preceding line. As when changing lines, **d 10** [RETURN] and **11 dd** are equivalent and can be used interchangeably.

Remember that when you use **c** or **d** and a line-oriented cursor control command such as **+**, **−**, **G**, **j**, **k**, **H**, **L**, or [RETURN], the current cursor line is replaced or deleted. On the other hand, when text-oriented cursor control commands such as **w** or **W**, **b** or **B**, and such, or text search commands (**/** or **?**) are used, the change or deletion begins at the present cursor character location and affects text from the current character to the new cursor location.

When using **c** or **d** with **/** or **?**[1], text is changed as follows:

- **Forward search** (**/**): Text is changed starting with current cursor character up to but not including the first character in the text string that matches the search expression.
- **Backwards search** (**?**): Text is changed starting with the first character in the text string that matches the search expression up to but not including the original cursor character.

Here are several commands for changing, deleting, and swapping lines of text in a file. They are defined earlier in this chapter, but are shown together here to illustrate various approaches to a given task.

[1] The / and ? commands are discussed in greater detail in the Pattern Searches section later in this chapter.

Line-Change Commands

Command	Action Taken
c RETURN, **c +**, or **2cc**	Replace current cursor line and the line that follows with new text until ESC is pressed.
c *n* RETURN or **c** *n* **+**	Replace current cursor line and the *n* lines that follow with new text until ESC is pressed.
*n***cc**	Replace *n* lines starting with the current cursor line with new text until ESC is pressed.
c −	Replace current cursor line and the line that precedes it with new text until ESC is pressed.
c *n* **−**	Replace current cursor line and the *n* lines that precede it with new text until ESC is pressed.

Delete or Swap Lines Commands

Command	Action Taken
dd	Delete current cursor line.
*n***dd**	Delete *n* lines beginning at cursor line.
ddp	Swap cursor line and the line that follows it.

Pattern Searches

You will frequently want to search through a file for a certain (often a misspelled) word or expression during normal edits. *Vi* provides a forward/backwards pattern search capability that is very useful for locating a certain phrase or word in a file, finding a misspelled word, or locating a line in a program. You provide the phrase or word or an excerpt from the line, and *vi* does the work. If a word is misspelled, provide the word as misspelled, let *vi* find it, then make the needed correction.

You can also do repetitive searches for the same pattern so that you can make a change based on the location of the pattern, then repeat the change for all or some of the other occurrences of the same pattern in the file. Here is how it is done.

Forward Searches

To search forward in a file for a certain text pattern, use the forward search command:

/pattern

where *pattern* is a series of characters that match a text word or phrase that occurs in the file. When you press [RETURN], *vi* searches the file beginning at the current cursor location and moving in the forward (toward end of file) direction until it finds the pattern or encounters end-of-file. If end-of-file occurs before the pattern is found, the search wraps to beginning of file and continues until the pattern is found or the current cursor line is reached. If the pattern cannot be found, the message:

```
Pattern not found
```

is displayed and the cursor is returned to its original position prior to the search command.

If the pattern exists in the file, the cursor stops at the first character in the first detected occurrence of the pattern. You can then make text changes or choose not to.

Searching Backwards in a File

You can search backwards in a file from the current cursor position by using the **?** command instead of **/**. Thus,

?pattern

searches backwards for the first occurrence of *pattern*. If beginning-of-file is encountered before *pattern* is found, the search wraps to end-of-file and continues until it returns full-circle to the current cursor line if the pattern is not found. As before, *vi* displays a `Pattern not found` message if the pattern is not present in the file or stops the cursor at the first character in the specified *pattern* if it is found.

Repeating the Search

Sometimes you may need to search for every occurrence of the pattern in the file. Obviously, it makes little sense to have to retype the pattern each time you want to continue to the next occurrence. You can use the **n** command to find the next occurrence without using RETURN. When you press **n**, *vi* immediately resumes the forward search from the **current cursor position** (not from the previous pattern location).

You can reverse the direction of the search for the next occurrence of the pattern by using the **N** command instead of **n**. Thus, if your last search for *pattern* was in the forward direction, **n** searches forward from current cursor position for the next occurrence of *pattern* while **N** searches backwards. Conversely, if your last search was in the backwards direction, **n** searches backwards while **N** searches forward for the next occurrence of the same *pattern*.

Defining the Pattern

The *pattern* associated with a forward or backwards search initiated by the **/** or **?** command is most commonly a simple string of characters exactly matching the text or word of interest. However, it is not limited to only simple strings. Any valid regular expression (RE) recognized by *vi* and *ex* can be used for *pattern*. Using regular expressions in this manner harnesses some of the massive processing power implemented on HP-UX, thus providing a great deal of flexibility. This means that REs can become quite complex, depending on their use. Refer to the tutorial on regular expressions earlier in this volume for a detailed discussion about how to use them. If you are a beginner, you can easily use alphanumeric characters to form search patterns for finding text. If you use non alpha-numerics, remember that some of them are used to represent other characters or combinations of characters and their use requires special care.

Shifting Lines Horizontally Left or Right

vi has a shift left/right command that moves the entire current line (or specified number of lines starting with the current line) right (>> command) or left (<< command) *shiftwidth* columns. The default value for *shiftwidth* is **8**, but it can be altered by using the **:set shiftwidth=***n* command where *n* is the number of columns to shift.

The shift-right (>>) and shift-left (<<) commands are useful on those occasions where you may need to move the left margin of a text block or paragraph right or left from the left margin or its current position. One method commonly employed by casual users is to insert or delete tabs or spaces at the beginning of each line. However, this can be cumbersome when a large number of lines are being shifted.

Consider the following lines of text:

```
These lines are about fifty characters in length.
They don't take a lot of space but they leave a
wide margin at the right-hand side.  If the lines
are shifted over, they move closer to the center.
```

By placing the cursor on the first line and typing **4>>** (shift four lines one shiftwidth to the right), they move one shiftwidth (8 columns) to the right.

```
        These lines are about fifty characters in length.
        They don't take a lot of space but they leave a
        wide margin at the right-hand side.  If the lines
        are shifted over, they move closer to the center.
```

Note that the cursor remains on the same line so you can easily press **.** to repeat the operation if you want to shift right again. To cancel the most recent shift, press **u** (undo). To shift four lines one shiftwidth to the left, type **4<<**.

Shiftwidth is discussed in Chapter 12 entitled Configuring the Vi/Ex Editor.

Automatic Indenting

When using structured programming languages such as C or Pascal, it is desirable to change indentation for each level in hierarchical source code structures. The *vi*/*ex* editor provides an **autoindent** option that supports this feature. It is described in detail under the **autoindent** heading in the chapter entitled Configuring the Vi/Ex Editor.

vi and *ex* are normally configured with autoindent disabled (default) unless a user configuration file named *.exrc* exists in a given user's home directory (see Chapter 12 for information about configuration files). To enable autoindent without disturbing other aspects of your editing session, simply type:

```
:set ai RETURN
```

To disable autoindent at any time during the session, type:

```
:set noai RETURN
```

Using Automatic Indentation

When autoindent is enabled, *vi* identifies the position of the first visible character on the current cursor line and sets that column number as the current indent value. Whenever a new line is created while in insert mode (**o**pen or **O**pen create a new line immediately; insert/append/change/substitute create a new line whenever RETURN is pressed or whenever wrapmargin forces a new line if it is set), the new line automatically begins at the current indent.

Changing Current Indent

The current indent is easy to change. To increase the indent on a new line, use spaces or tabs to obtain the appropriate indent. To decrease the indent, use the special command CTRL-D (end-of-file character) which moves the current indent left by **shiftwidth** characters or to the left margin, whichever occurs first.

Good program structure also includes comment lines and other features to promote readability and ease of interpretation. It is often desirable to provide a single line (such as a comment) at the left margin, then resume normal indent for succeeding lines of program code. This is easily accomplished with a slightly different technique for moving the current margin to the left. Assuming that you just finished a line starting at the current indent and pressed RETURN to start the next line (or perhaps twice to obtain a blank line):

- Press the circumflex key (^) as the first character on the new line. The character is printed on the screen by *vi*.
- Press [CTRL]-[D] to select extreme left margin for one line only.
- The circumflex character disappears from the screen and the cursor is moved to the extreme left column on that line.
- Type the new line as desired, starting at the left column and inserting desired white space, if any, on the left before visible text.
- When the line is finished, press [RETURN]. The cursor appears below the left margin of the previous line, not the left margin of the line that was just typed.
- Use the same procedure to type another line at the left margin, continue with the current margin, or change it right or left, as desired.

In Summary

- Use space or tab characters to move autoindent margin right.
- Use [CTRL]-[D] (must be first character typed on new line) to move margin left by one shiftwidth.
- Use [^] followed by [CTRL]-[D] to type a single line at left margin without changing current indent value.

Use of Tabs

Autoindent uses tab characters where possible to conserve the total character count in the file being edited. Some applications may require that all leading whitespace be only spaces. In such instances, you must either edit the file with autoindent disabled, or process the file using the *expand* command which converts tabs into spaces. To expand tabs in a file, use a command similar to:

```
expand filename >new_filename
```

which replaces tabs with spaces such that text is lined up at intervals of eight characters starting at the left. Other forms of the command can be used to specify tab column positions other than every eight. Another technique in specifying options provides the ability to specify that tabs be assigned to stop on certain column numbers in each line. Refer to the *expand*(1) manual entry in the *HP-UX Reference* for more information.

Intermediate Editing: Using Text Objects

5

One of the great strengths of *vi* is its ability to handle a variety of text objects such as words, sentences, paragraphs, sections, and such, and its ability to allow each user to independently define certain objects such as sections by using *vi* configuration commands. Learning about and understanding these text objects and how they are used is a critical key to accessing and using the true power that is available from *vi* for the experienced user. Text objects were introduced in Chapters 3 and 4 with limited discussion. This chapter explains them in greater depth so that more experienced users can make better use of them.

Common Text Objects

Word: w or W

Word objects are treated in two ways: Words and words. When the uppercase **W** is used, word boundaries are delimited **only** by white space which is defined by any one or more or a combination of blank, tab, or newline (end-of-line sequence) characters (see Glossary for more information about white-space characters). When the lowercase **w** is used, word boundaries are delimited by white space or detection of any other non-alphanumeric character (except underscore). A sequence of one or more contiguous non-alphanumeric characters (other than white space and underscore) is treated as a single word. Thus the expression:

```
This, at least, is a sentence.
```

is treated as nine words or six Words (the lowercase **w** version treats each comma and period as a separate word). On the other hand, the expression:

```
$%one_word#&(@
```

is treated as one Word or three words: `$%`, `one_word`, and `#&(@`.

Line: ^ or $

The current line and the current position within the current line form the basic foundation from which *vi* operations are referenced. Two characters are used in *vi* and *ex* commands that represent the beginning and end of the current line:

^ When used in a regular expression, this character represents the position that would be occupied by a character if it were inserted in front of the first character in the line (left screen margin).

When used as a cursor control command, the cursor moves to the first visible (non-blank) character in the line.

$ When used in a regular expression, this character represents the position that would be occupied by a character if it were appended after the last character in the line.

When used as a command, the cursor moves to the last character (visible or invisible) at the end of the line.

Note

End-of-line refers to the end of the line as it exists in the buffer file, not as it is displayed on the terminal screen. If the line is longer than the width of the display screen, beginning and end of line occur on separate lines of the display.

Sentence: (or)

vi accommodates two types of sentences:

- A group of words terminated by ., !, or ? followed by at least two spaces (blanks) or end-of-line. This defines a sentence as commonly used in normal spoken language, and is consistent with standard typing practice. If the ., !, or ? is followed by fewer than two spaces (unless at end-of-line), it is not treated as an end-of-sentence condition.
- For the Lisp programming language, a sentence is a valid Lisp s-expression, provided the −l (Lisp) option was specified as part of the HP-UX *vi* command at the beginning of the session as discussed in Chapter 2, or the **:set lisp** command is included in an EXINIT variable or *.exrc* file or executed as a command from the editor.

Paragraph: { or }

A paragraph is a group of one or more sentences that is bounded before and after by:

- One or more blank lines, or
- Any paragraph macro among:
  - The default paragraph and section macros defined by *vi*, or
  - An alternate set of macros defined by a **:set paragraphs** and/or **:set sections** command.
  - But not both.

New paragraphs begin at a blank line, defined paragraph macro, or defined section macro.

Unless redefined by a **:set paragraphs** and/or **set sections** command, the following default paragraph macros from various *nroff/troff* macro packages are recognized by *vi*:

.IP .LP .PP .QP .P .LI .bp

To add new macros or change the current list of recognized paragraph macros, the entire string must be redefined. Refer to the chapter entitled Configuring the Vi/Ex Editor for procedures.

Section: [[or]]

A section is a block of text bounded by a section macro or end-of-file where the section macro marks the beginning of a new section. Recognized section macros are defined by the **:set sections** command.

vi recognizes the following default section macros that originate from various *nroff/troff* macro packages:

.NH .SH .H .HU

To add new macros or change the current recognized section macros, the entire string must be redefined. Refer to the chapter entitled Configuring the Vi/Ex Editor for procedures.

User-Defined Text Objects

In addition to the preceding text objects recognized by *vi*, you can define any other text object by specifying the text lying between the current cursor position and any standard cursor move command including text markers. User-defined text objects fall into two general categories:

- Line-oriented objects identified by beginning and ending line, and
- Exact-position-oriented objects such as all characters between two markers.

Note

Text pattern searches (/ and ? commands) cannot be used in certain situations when defining a text object.

Text Markers Within a File

File markers are a locating device used by *vi* and other editors to accurately pinpoint a specific location in a file. *vi* file markers can be used to locate a specific character in the file or the beginning of the line containing the marked character, depending on the type of command used with the marker. *vi* file markers are not stored as part of the file, so they are lost at the end of the editing session.

Up to 26 file markers can be specified in any given file during the editing session. Each marker is given a lowercase single-character name in the range **a** through **z**.

Creating Markers

To mark a location in the file, use any normal combination of screen and/or cursor control commands to move the cursor to the desired location in the file. Once the cursor is in the correct location, type the command:

`m`*marker_name*

where **m** is the "mark file location" command and *marker_name* is a single lowercase character in the range **a** through **z**. If an illegal character is specified for *marker_name*, the command is ignored and the editor sends a beep sequence to the terminal to signal the error.

Up to 26 simultaneous locations can be marked in the file during any given editing session. If a new marker command is given and the marker name is the same as an existing marker, the previous marker is cancelled then redefined for the new location.

Using Markers for Cursor Control

One common use for markers is as a convenient means for moving quickly between arbitrary locations in a large file. For such uses, marked text locations can be reached by two methods. The first command form:

`` `marker_name ``

precedes the *marker_name* with an accent grave (also variously called a backwards single quote or other similar names) moves the cursor to the exact **character** in the file that is identified by the *marker_name* provided as part of the command.

On the other hand, using a single quote (apostrophe) as follows:

`'marker_name`

moves the cursor to the beginning (first visible non-blank character) of the line containing the marked character.

This subtle but important difference between forms is very useful in certain situations when performing editing text objects defined by marker-specified boundaries.

Using Markers for Text Object Operations

Markers can be used to define text objects being manipulated by a **d**elete, **c**hange, or **y**ank command. In general, the text object is bounded by the cursor position at the time the command is given and the location of the marker specified in the command sequence or by the lines containing each, depending on the cursor move command associated with the marker name.

Note

If a line or characer associated with a file marker is deleted, the marker definition is also cancelled at the same time.

As indicated in earlier chapters on basic editing, the text characters contained within the text object depend on whether the cursor position before the move operation is at the beginning or end of the object in the file. Here is the general structure of the editing commands as they are used in conjunction with file markers:

1. Move cursor to beginning or end of text object being manipulated.
2. Specify operation type:
    - **d** to delete entire text object,
    - **d** preceded by buffer name to delete text object into named buffer (buffers are described in Chapter 6),
    - **c** to change text object to replacement text,
    - **y** to yank text object into default buffer,
    - **y** preceded by buffer name to yank text object into named buffer,
3. Specify other boundary of text object being manipulated.
    - *'marker_name* sets boundaries on marked character and current cursor position. Text object includes first boundary character in file and all text up to second boundary character, but does not include the second boundary character.
    - *'marker_name* sets boundaries to include entire line containing marker and entire line containing cursor character, regardless of their relative positions in the file.

Examples

The flexibility of text objects, especially when dealing with sentences, paragraphs, sections, and markers, make it difficult to provide any quick useful examples involving large text blocks. However, here are a few examples of commands that reference text objects so that you can readily become quite proficient in their use with a little practice.

In the following table, marker **a** is assumed to precede marker **b** in the file.

Command	Action Taken
'ad'b	Remove all characters starting with the character marked by marker **a** up to but not including the character marked by marker **b**.
'ad'b	Remove all lines starting with the line containing marker **a** and continuing through the line containing marker **b**.
'adG	Remove all lines starting with the line containing marker **a** and continuing through the last line in the file.
'ad'b	Remove all lines starting with the line containing marker **a** and continuing through the character preceding marker **b**.
'ad'b	Remove all text starting at marker **a** and continuing through the entire line containing marker **b**.

The list of examples could be expanded to hundreds of pages, but we must stop somewhere. In this and the preceding chapter, you have the tools to learn all you need to know about the types of operations described.

Intermediate Editing: Copying and Moving Blocks of Text 6

vi/*ex* provides three basic ways for moving and copying blocks of text from one area in a file to another:

- Delete or yank text into the default (unnamed) buffer or any of the 26 named buffers maintained by *vi*, then use the put command to copy the buffer contents into the same or one or more different locations in the file. A text object of any size can be handled in this manner.

  Line numbers, line addresses, word, sentence, paragraph, section, markers and other commands can be used to determine what text block is placed in the buffer by the delete or yank command. If named buffers are used and a second file is opened from the first editing session, you can use this technique to copy or move text from one file to another using the techniques described in Chapter 8 for editing two files simultaneously by switching between files.

- Use the *ex* move or copy command to transfer or copy one or more lines to a different location in the file. Only full lines can be handled in this way. This technique is restricted to move/copy operations within the current file.

- Use the *ex* write file command to copy one or more lines in the file into a second mass storage file, delete the copied lines from the file if desired, then copy the new file back into the file being edited at the new location. This technique is not frequently used because it is somewhat more cumbersome, but it does reduce the risk of losing the contents of the unnamed buffer during a move due to an accidental incorrect keystroke.

  This method is also commonly used for copying blocks of text from one file to another.

Most text copy and move operations during *vi* sessions, especially those that involve moving or copying text to another location in the current file, use buffers as an intermediate storage facility during the copy or move. Some users prefer to use the *ex* **copy** and **move** commands to accomplish the same thing. Use of external files as a temporary resource for accomplishing copies and moves is most common when the transfer is between files, especially when a single text block is being copied into several other files.

Using Buffers

vi maintains 27 buffers that are always available for use when copying or moving blocks of text. They are:

- The default, unnamed buffer,
- 26 named buffers **a** though **z**.

The most commonly used buffer is the default buffer which is somtimes referred to as the unnamed buffer. Whenever a delete or yank operation is performed, the deleted or yanked text is copied into the default buffer (unless another buffer name is specified). It can then be placed elsewhere or replaced in its original position by using the **put** (**p** or **P**) command. However, the contents of the default buffer are maintained only until the next text modification command is executed. Thus, if you delete a block of text (causing it to be placed in the default buffer), move elsewhere and execute another text modification command, then move again and try to place the buffer contents in your new location in the file, you will discover that the buffer text was destroyed by the command executed since the deletion. The buffer is destroyed even if the change is not a deletion or yank (for example, an insert or append).

The contents of the named buffers, **a** through **z**, remain intact for the entire time that *vi* is running except when a new delete or yank to that named buffer overwrites the buffer contents. All buffers, both named and unnamed, are dismantled when *vi* terminates; not at the end of editing the current file. This means that information can be copied into named buffers when editing multiple files, then the contents of those buffers can be placed in other files being edited as part of the same session.[1]

Note

The default (unnamed) buffer contents are destroyed at the end of a file edit, even if several files are being edited in a single session. If you need to copy data between files using buffers, use named buffers instead to preserve contents until the appropriate file is open.

---

[1] The **:rewind** command can be used to move back to the first file in a group of files, providing useful flexibility in manipulating data between files. Techniques used when editing multiple files in a single session are described in greater detail in Chapter 8. The general techniques of deleting or yanking to a buffer then placing back in a file apply to both editing within a file and editing multiple files.

Naming and Filling the Target Buffer

vi/ex predefines the named buffers **a** through **z**. However, you must specify the name of the buffer being used for a particular operation unless you are using the default unnamed buffer. You recall that, when we discussed file markers, the reverse single quote or accent grave (`) followed by a marker name is used to move the cursor to the exact **character** location of the specified marker. The standard single quote ('), also called apostrophe or acute accent, followed by a marker name moves the cursor to the beginning of the **line** containing the specified marker. You may also have noticed that the available marker names are the same as the available buffer names, except that the buffer names are accessed by using a double quote before the buffer name which is specified prior to the delete or yank command in the command line.

Here are some examples of how text is copied into a named buffer:

Command	Action	Buffer
"a6dd	Delete 6 lines starting with current line.	**a**
"r3yy	Yank 3 lines starting with current line	**r**
"xd4)	Delete 4 sentences starting at current location	**x**
"hdG	Delete all text from current line through end of file	**h**
"b3x	Delete 3 characters starting at current position	**b**
"gy/*text* RETURN	Yank from current position to but not including *text*	**g**

Note

Named buffers are named **a** through **z**, as are the 26 possible marker names. However, text buffers and editor file buffer markers are completely unrelated and independent of each other.

Appending Text to Buffers

Text can be appended to an existing buffer instead of replacing any current buffer contents. Simply use the uppercase buffer names **A** through **Z** instead of the corresponding lowercase buffer names **a** through **z**.

Retrieving Text from Buffers

Anytime prior to end of session, data can be retrieved from a named buffer and placed in text relative to the current cursor position. To place the data, use the form:

"⟨*buffer_name*⟩p

to place text after current character or line, or

"⟨*buffer_name*⟩P

to place text before current character or line. As in normal yank/delete and put operations using the default buffer as described in various locations in Chapter 4, if the buffer contents was originally yanked or deleted relative to current cursor positiion in a line (character, word, sentence, section, etc.) the buffer text is placed in the current line starting relative to the current cursor position within the line. If, on the other hand, the yank/delete operation was performed on full lines (*n*dd, *n*yy, etc.), the **put** command places text before or after the current line.

Thus, using the previous yank/delete examples:

Command	Buffer	Text Size and Placement
"a6p	**a**	Six lines previously deleted after current line.
"r3P	**r**	Three lines previously yanked before current line.
"x)	**x**	4 sentences starting after current character.
"hP	**h**	Lines to former end-of-file before current line.
"bP	**b**	3 characters before current character.

Executing a Buffer as an Editor Command

Many commands, especially search-and-replace commands, can involve tedious typing with risk of mistakes, especially if you are a bit clumsy with the keyboard on occasion, or perhaps a casual or inexperienced user.

vi has a rarely documented but very useful feature that can be used in conjunction with the yank command to save much retyping if you are doing complex *ex* operations that require a lot of typing. The procedure is simple:

- Use a yank-line (**Y**) command to yank the entire current cursor line (which must be a valid *ex* command including the colon at the beginning) into a named buffer. For example, to use buffer **h**, the command is:

  "hY

- Use the *vi* **@** command followed by the buffer name to execute the buffer contents. For example, to execute the contents of buffer **h** that was previously yanked, type:

  @h

  No RETURN is necessary in either case because both commands are processed by the *vi* command-mode interpreter.

  The editor places the command in the buffer on the bottom line of the display just as it would appear if you had typed it in *ex* command mode, then executes the command. You will note that if the command was a substitute command, the substitution is made on the regular expression part of the command as well as anywhere else in the text file unless the address range was restricted to only part of the file.

  The substitute command is discussed in greater detail in the next chapter.

When using this technique, it is usually best to place lines being yanked at the beginning or end of the buffer file so that they can be easily deleted at the end of the session before writing the file back to permanent storage.

Using Ex Commands to Copy or Move Text

The *ex* commands, **copy**, **move yank**, and **put** can also be used to copy or move text directly or through named buffers. Procedures are similar to using *vi* methods with some obvious variations. Use of each command is discussed in detail in Chapter 10.

Using Files to Copy or Move Text

External mass storage files can also be used to copy all or part of a file to another location in the file or to other files. To move text, the lines of interest are written to a file then the lines are deleted from the current workfile. To copy text, the lines are written to a file but not deleted. The external file can then be read back into the file being edited or into another file.

For more information about using the read and write commands, **:r** and **:w**, refer to Chapter 9 which discusses file manipulation techniques.

Intermediate Editing: Search and Replace Operations 7

Search-and-replace capabilities are an important feature in any useful editor program. The search-and-replace features in this editor are accessible both from *vi* and *ex* as *ex*-mode commands. This chapter describes several techniques for using search-and-replace to solve a variety of editing problems.

The simplest form of search-and-replace is conducting a pattern search as described earlier in Basic Editing, then performing a character, word, or line replacement. The replacement can be done on other occurrences of the same pattern in the file by pressing [N] to find the next occurrence, then pressing [.] to repeat the previous change.

Obviously, this process becomes dull and tedious rather rapidly if a large number of changes are needed in the file. The *ex* capabilities addressed in this chapter can be used to automate this process greatly and save considerable time and effort. However, be wary because if you do not exercise adequate care in defining the string to be changed, you may get more changes than you really wanted. For example, changing every occurrence of *the* to *xyz* in a file also changes *Athena* to *Axyzna*; probably not what you would want.

Search-and-replace operations are most commonly performed using the substitute command in the *ex* command set with a global suffix on the command if the operation is to be performed more than once on any given line. The *ex* command set is accessed from *vi* command mode by typing a colon which switches *vi* to *ex*-mode operation for the duration of *ex* command execution.

Colon Commands

ex-mode (or external-mode) commands always begin with a colon when accessed from *vi*, and are therefore often referred to as colon commands. When forming a colon command, the first character, the colon, is followed by a command sequence that can be any legitimate *ex* command (except for certain cases when program bugs or other considerations require a change to *ex* by using the **Q** command). *ex* commands are discussed in detail Chapter 10. This chapter deals primarily with the substitute command and related items necessary to form a valid search-and-replace command expression. To execute a completed command, press [RETURN] or [ESC] after typing it ([RETURN] is most commonly used and preferred, but [ESC] also works).

Fixing Mistakes

If you make an error while typing a colon command, use [BACK SPACE] to move the cursor left to the appropriate position, then retype the rest of the command. As with normal *vi* operation, characters are not erased from the screen as you move the cursor left, but they are removed from the *vi*/*ex* command buffer. Hence, any extra characters that are not obliterated by retyping are ignored (you will notice that they disappear from the bottom line of the display as soon as you press [RETURN] or [ESC]).

If you make a mistake early in the line and prefer to retype the complete line, press the KILL (line-erase character, usually [CTRL]-[U], which immediately moves the cursor to the first character following the colon so you can type a new line.

Aborting the Command

If you type part of a colon command then decide you want to do something else instead, you can abort the command by pressing [BACK SPACE] several times (or press [CTRL]-[U] followed by [BACK SPACE]) to back the cursor up to the left margin past the colon. When the cursor passes the colon, the the editor abandons the command and returns the cursor to where it was prior to the aborted colon command. This same method can be used to abort a *vi* search command (/ or ?).

An easier method is to simply press the [BREAK] key. This method has the side-effect of setting the HP-UX *vi* command return status flag to FALSE when *vi* terminates, but unless you are operating in an unusual environment, using the BREAK key should present no discernable disadvantage.

Aborting After Execution Begins

You may discover, particularly when performing global operations on a very large file, that you gave an incorrect command (such as inadvertently pressing / or ? instead of :) or an inappropriate command, and need to abort it. Press [BREAK]. Command execution stops, and an error message is displayed:

```
Interrupt
```

If you accidentally type a search command (/ or ?) and interrupt the search with [BREAK], the cursor usually returns to its original position prior to the command, and the file remains unmodified by the command. If you interrupt a substitution part-way through the file or if the incorrect command runs to completion before being interrupted, you can use the **u** (undo) command to repair the damage and return to the pre-command state.

Should the screen be left in an unusable state (not likely but it can happen on occasion, simply press CTRL-L to redraw it.

Undoing Colon Commands

Like normal *vi* commands, the external-mode commands are also subject to the **u** command. If you discover that the change you made did not produce the desired effect, press **u** immediately before executing any other command. As usual, if any command is executed after the colon command, the undo option for that command is forever lost, and you must either use another command or set of commands to fix the error or abort the session (**:q!** command) and start over.

File Safety

Because certain complex *ex* commands can have a disastrous effect on a file if they are incorrectly formed and you forget to execute an **undo** before you execute the next command, it is a good idea to write your buffer file to permanent storage before performing a complex instruction. That way, if the command happens to demolish your file beyond use, you can easily abort by using **:q!** without overwriting the back-up, then use the *vi* command again to reopen the file. Use of write and quit commands is discussed in greater detail in Chapter 2.

Command Structure

The search-and-replace command consists of the following parts in the following sequence:

- A colon to identify the command as an *ex*-mode command
- A starting line number or address
- An ending line number or address
- A substitute command
- A regular expression that defines the search string to be identified
- A substitute string to replace the search string when found
- A global suffix if the operation is to be performed more than once on any lines containing two or more strings that match the search string expression.

Line Addresses

Colon search-and-replace commands start with a colon which tells *vi* to execute the *ex*-mode command (*ex* can also execute the same command except that the *ex* personality provides a colon prompt so it is not necessary to type a new colon, although if you do type a colon while in *ex* it is not treated as an error). The colon is then followed by zero, one, or two line addresses where:

- If no address is present, it implies that the operation is to be performed on the current line only.
- One address tells the editor to perform the operation on the addressed line only.
- When two line addresses are provided, the operation is performed on all lines starting with the first line addressed and continuing through the second addressed line. The first line address must precede the second address in the file. A comma separates the first and second line address.

Whitespace (space or tab character) is optional but rarely used before, between, and after line addresses.

All colon commands include a line address for a single line, a double line address for a group of contiguous lines, or an implied address when no specific address is included in the command. Here is a list of line address forms recognized by *ex*:

Recognized Colon Command Line Address Forms

Address	Corresponding Line
none	Current line only (implied address)
1	First line in file.
$	Last line in file.
.	Current line.
n	*n*th line in file.
.−*n*	*n*th line before current line.
.+*n*	*n*th line after current line.
%	Abbreviation for **1,$** which means every line in the file.

When two addresses are present to define starting and ending line numbers for the command, they must be separated with a comma (,) as shown in some of the following examples:

Examples of Colon Command Line Address Forms

Address	Corresponding Line
1	First line in file.
n	Line *n* in file.
.	Current line in file.
.−4	Fourth line before current line in file.
.+8	Eighth line after current line in file.
$	Last line in file.
g	All lines in file.
1,.	All lines from beginning of file to current line.
.,$	All lines from current line to end of file.
.,.+5	Current line through fifth following line.
1,.+5	First line in file through fifth line after current line.
.−10,.+5	Tenth preceding line through fifth following line.

Global searches (described next) can also be used to identify certain lines in the file in lieu of the address forms in the preceding list, as can file markers.

[1] Space characters after the colon and before and/or after the comma are optional but not normally used.

Global Searches

Suppose you are working on a large file such as a large computer program or text file and need to look at every line in the file that contains a certain word, program label reference, or operand name. Rather than using a cumbersome series of / or ? followed by **n** or **N** search sequences, you can print all occurrences of the desired text pattern with a simple command of the form:

:**g**/*text_pattern*/**p** RETURN

where *text_pattern* is any regular expression of the form described in the tutorial on regular expressions earlier in this volume that is compatible with *vi* and *ex*. The **g** command specifies that the search is to be made globally (on every line) throughout the file, and the **p** command specifies that the results are to be printed on the display screen. Experienced users will recognize that this command is very similar to the HP-UX *grep* command.

After the lines are printed to the screen, the message:

```
[Hit return to continue]
```

appears at the bottom of the screen. Press any typing key to restore the normal editor display.

Limited Searches

You can easily limit the search for a given expression to a certain part of the file by specifying the starting and ending line numbers. Here is the command form:

:*start_line*,*end_line* **g**/*text_pattern*/**p** RETURN

where *start_line* is any valid line number identifier that specifies the starting line and *end_line* specifies the the last line in the search space. Valid line specifiers can be the actual line number (**1** is the first line in the file, **$** specifies the last line, and **25** specifies line 25, etc.), line locations relative to current line, or any other form recognized by *ex*.

Displaying Tabs and other Control Characters

Suppose you need to determine whether and where any control characters might be hidden in a file. This can be particularly important when examining a computer program file as well as in many other circumstances.

The **l** command accomplishes this task quite handily with the form:

:*start_line*,*end_line*l RETURN

Any control characters contained within the specified file segment are displayed in "hat" format, "hat" being a common vernacular name among UNIX users for the circumflex character (^). Tabs are displayed as **^I**, and end-of-line is displayed as **$**. For example, consider the following rather innocent looking line of text:

```
If this looks like a simple sentence, look between the words
```

Placing the cursor anywhere on the line and executing the command:

:.l RETURN

reveals more than what meets the eye:

```
If this looks like a^Isimple sentence, look between the words  ^I  $
```

showing two hidden tabs plus several spaces at the end of the line. Likewise, a command of the form:

:.,.+101 RETURN

displays all control characters in the current plus the 10 following lines.

After listing the lines, press any key to restore the normal editor display.

Splitting Lines

One very perplexing problem for many *vi* users lies in how to split lines during a pattern search-and-replacement operation. In other words, when a certain pattern is found, how can an end-of-line be placed near that location so that multiple lines are formed. Likewise, how can a pair of lines be combined in a global search-and-replace sequence?

Switch to Ex

It is impossible to accomplish either of these feats from *vi* because the *vi* command interpreter cannot handle multiple-line command input. However, the task is much easier when using *ex*. To switch from *vi* to *ex*, press **Q** (SHIFT-Q). *ex* then responds with the usual colon prompt.

Forming the Command

Like any normal search-and-replace command, the colon prompt is followed by typing an address or pair of addresses followed by the **s** (substitute) command. A regular expression followed by replacement text forms the substitution part of the command, and any flags or options are added at the end of the command.

For illustration, let us use the following paragraph as original text:

```
This paragraph is only one of the many possible demonstrations
of the substitute command for splitting lines.  However, the
result may make you writhe in pain or laughter as your friends
watch the scene.
```

With the cursor on the first line of the paragraph, press **Q** to get the colon prompt, then type the following command:

```
:.,.+3s/the /the\
/g
```

Now for the explanation. The colon prompt was provided by *ex*. The **.,.+3** specifies the current and three following lines. **s** says substitute the replacement text for the text pattern defined by the first expression. The first expression is identified by the slash characters before and after. The second expression consists of **the** followed by a backslash (\) which is in turn followed by RETURN and another slash, meaning that the replacement ends with an end-of-line (the \ at the end of the first line in the command escapes the newline character from interpretation by the editor as end-of-command). The **g** option tells *ex* to make the substitution for every occurrence of the first expression throughout the specified body of text even when it occurs more than once on any given line.

Executing the command yields the following displayed line:

```
scene.
```

Now, change back to *vi* by typing `vi` RETURN. The previous line appears at the top of the screen. Scroll the text down to reveal the following:

```
This paragraph is only one of the
many possible demonstrations
of the
substitute command for splitting lines.  However, the
result may make you writhe
in pain or laughter as your friends
watch the
scene.
```

As expected, every occurrence of **the** followed by a space was replaced with **the** and a newline sequence. Even **writhe** was subjected to the same treatment as you would expect since no restrictions were placed on the text preceding **the.**

Unfortunately, regular expressions cannot have newlines escaped into them, so it is not possible to reverse the process and join lines using global search-and-replace operations.

Switch Back to Vi

After the operation is completed, you can switch back to *vi.* Type the *ex* command:

`vi` RETURN

vi is then reactivated, with the current line from the last *ex* operation at the top of the screen. You can use scrolling, *n*G, or any other suitable screen control command to move to a particular location in the file, or use any other appropriate command to continue editing.

Another Example

Using the first sentence from the previous example, let us change a few of the ASCII space characters to tab characters as follows:

```
This paragraph is     only one of the many     possible
demonstrations of the         substitute       command for
splitting    lines.
```

Now, using the previous procedure, place the cursor on the first line of the three, press **Q** to switch to *ex*, then execute the following command:

```
:.,.+2s/[space tab]/\
/g RETURN
```

where the regular expression [*space tab*] contains a space and tab character; that is, [⟨*space*⟩⟨*tab*⟩]. This tells the editor to search for any space or tab character and replace it with the replacement expression which is an end-of-line preceded by an escape character to protect it from being devoured by the editor commands interpreter. The **g** option at the end tells the editor to perform the substitution for all matches throughout each line in the body of text being searched. After executing the **vi** command to restore the *vi* personality, the sentence looks like this:

```
This
paragraph
is
only
one
of
the
many
possible
demonstrations
of
the
substitute
command
for
splitting
lines.
```

There is a defect in this example in that if two or more spaces and/or tabs appear in succession, each is converted to a newline which results in blank lines in the output. This is easily solved by the following command:

```
:.,.+2s/[space tab][space tab]*/\
/g RETURN
```

which tells the editor to replace any space or tab followed by **zero** or more spaces and/or tabs with a newline (treating the series of one or more characters as a single entity), and perform the substitution globally across the line. The asterisk after the second closing square bracket tells the regular expression evaluator to look for zero or more of the previous character which can be either a space or a tab as specified by the enclosing square brackets. If no tabs exist in the file, the search expression can be simply two spaces followed by an asterisk. The technique of searching for spaces and/or tabs is used because HP-UX and other similar operating systems use the term "blank" to mean either a space character or a tab character.

Double-Spacing Text

To double space text by placing a double newline at the end of each line, use the command:

```
:%s/$/\
/g RETURN
```

Strip Unneeded Blanks

It is often desirable to remove unneeded blanks (space and/or tab characters) at the end of every line in a text file in order to conserve disk or tape storage space. This is easily accomplished by using the following command:

```
:%s/[space tab]*$// RETURN
```

where *space tab* is a single space and a single tab character (or vice-versa) placed between square brackets. The asterisk after the closing bracket and the dollar currency symbol after the asterisk tell the editor to search for zero or more blanks and/or tabs at the end of every line in the file and replace them with nothing. Even blank lines that may contain invisible blanks are trimmed to empty strings by this command.

Save Time by Executing a Buffer

One of the frustrating problems of working with complex substitution commands, especially if the regular expressions used involve subexpressions (subexpressions are described in detail with several examples in the *sed* tutorial elsewhere in this volume). You can easily alter, fix, or modify a command and executed it over and over without retyping the entire command by using buffers.

For example, suppose you made a minor mistake when typing a substitute command which created an unwanted result. You can easily use the undo command to fix the damage and restore the original to its state before the command was executed, but you are now faced with retyping the whole command, and you don't have the previously typed line to use as a reference. There is an easier way.

Fixing Errors in Commands

First, type the command **on the last line in the file** instead of at the bottom of the screen. This is done by typing **Go** from *vi* while in command mode. The command moves the cursor to the end of the file (set a marker at your old location as described in Chapter 5 if you need to return to your previous location later), then opens a new line after the existing file.

Now, type the colon command, exactly as you normally would when using the *ex* command mode. When the command is fully typed, press ESC, then examine the command to make sure it is correct.

When you have a fully formed command, yank the line (cursor is on the line containing the colon command) using the technique described in Chapter 6:

"⟨*buffer_name*⟩**Y**

where ⟨*buffer_name*⟩ is the name of any buffer, **a** through **z**. The line is now in the buffer you have specified.

To execute the buffer, type two characters:

@⟨*buffer_name*⟩

The command appears at the bottom of the screen, and is immediately executed. Make sure the changes are correct before proceding. If they are not what you wanted, press **u** to undo the changes, then press **G** to return to the line containing the command, edit as needed, then repeat the procedure.

When you are finished, delete the line from the file.

Forming Multiple Substitute Commands

In complex edits, you will likely use many substitute commands. You can easily take a previous command on the last line of the file, modify it as needed, yank it to a buffer for execution, then repeat the process as required. Again, as before, delete the command line from the file before you terminate the edit.

Intermediate Editing: Editing Multiple Files

8

As you gain experience, you will likely encounter times when you need to edit several files in a single session. You may also need to edit two files simultaneously with the ability to:

- Arbitrarily switch back and forth between the two files while performing similar operations on each file, or
- Move text back and forth between two or more files without terminating the edit.

This chapter describes how to do various types of edits involving multiple files.

Editing Multiple Files in Succession

The most common type of multiple-file editing usually involves normal editing on two or more files in succession. The procedure is simple and very straight-forward.

Opening the Session

As in any editing session, start by executing the *vi* command. The only difference between this session and a single-file edit is that you specify more than one file to be edited. There are several ways to do this. For example, to edit files *one*, *two*, and *three* in the current directory, execute the following command:

```
vi one two three [RETURN]
```

vi opens file *one* and, if it exists, displays the beginning part of the file on the display screen. If the file does not exist, the `new file` message explained in an earlier chapter is displayed.

Edit the first file as if it were the only file being edited. You can close the edit on that file by using a write (**:w**) or terminate (**ZZ**) command. If you use the **:w** command, a message at the bottom of the screen shows the filename and number of lines and bytes in the file as follows:

```
"myfile" 179 lines, 6511 characters
```

If you use **ZZ**, the file name and size message is displayed briefly while the file is being written, then it is replaced by an new message indicating that you still have two files to edit:

```
2 more files to edit
```

In either case, to proceed to the next file, use the "next file" command:

```
:n [RETURN]
```

vi terminates the edit on the first file and opens the second file specified in the original HP-UX command that started the session. Edit and close the second file the same way as you did with the first file.

When the third file is opened, proceed as usual, and terminate as usual. Upon termination of the last file in the series, a **:wq** or **ZZ** command closes the file, ends the session, and returns to the shell process from which the session started. The shell then displays a new shell prompt on the terminal display screen.

Using Buffers in Multi-File Edits

When using this method to edit multiple files, **named** buffers are preserved between files. Thus you can yank or delete text into a named buffer as described in Chapter 6 while editing one file, then copy (**p**ut or **P**ut) the named buffer contents into a later file in the series. The contents of the unnamed buffer cannot be transported between files, but the last command is remembered so you can use the dot (**.**) command from file to file without losing the last operation.

Going Back to the First File

Sometimes, especially on long, complex edits, you may want to make additional changes on files already edited. You can reset the file pointer to the first file while editing any other file (but before you close the last file) by executing the command:

```
:rewind
```

This command closes the current file (if autowrite is set) then reopens the first file for editing. If autowrite is not set, execute a **:w** command then the **:rewind** command.

An alternate form of the rewind command can be used to abort the current file and immediately reopen the first file:

```
:rewind!
```

Using Shell Characters in Filenames

When specifying filenames in the *vi* command line, all normal shell special characters (sometimes called metacharacters or wild-card characters) such as * can be used. For example, suppose you have several files, each containing a chapter in a book project and you need to make some minor corrections in each chapter. If each file was named *chap1*, *chap2*, *chap3*, etc. the easiest way to specify the files being edited would be:

```
vi chap* RETURN
```

or even:

```
vi c* RETURN
```

if no other files in the directory started with *c*. You can use any legitimate expression that the shell can correctly interpret when specifying file names. The use of shell special characters and their expansion is discussed in any good textbook on the UNIX system and in various HP-UX manuals that address the Bourne, C, and Korn shells and their use.

Editing Two Files Simultaneously

Situations occasionally arise in computer programming and technical composition when it is useful to be able to open two files simultaneously and perform editing operations on both files with the ability to switch between files without losing the contents of buffers or memory of the last editing operation. Such a capability enables you to conveniently:

- Delete or yank text from one file and insert it in the other and/or vice-versa,
- Perform identical changes, one pair at a time, on both files.
- Search for an expression through one file using **/** and **?** followed by **n** or **N**, then switch files and continue in the other file.

vi does not have the ability to handle multiple files simultaneously, but it does allow you to switch between two files being edited without losing memory of the last operation including yanked or deleted text using named buffers. Thus you can switch to the other file and repeat the last change (using the **.** command) or put yanked or deleted text in the chosen location in the other file. This capability works best when the **autowrite** option is set as discussed in Chapter 10: *ex* Commands.

Opening the Files

To start editing the two files *file1* and *file2*, open *file1* with the command:

```
vi file1 RETURN
```

This command opens *file1* and, if it already exists, copies the file into the *vi* workfile buffer. Any editing operations performed at this time are placed in the buffer. The buffer must be written to permanent storage before the second file can be opened. If the **autowrite** option is set, this is done automatically. If not, the **:w** command must be used before opening the second file.

To open the second file, use the external-mode command **:e** as follows:

```
:e file2 RETURN
```

vi then writes the buffer to permanent storage if **autowrite** is set (or clears the buffer if it has been written already or has not been altered since it was opened), then copies *file2* into the buffer area if *file2* exists or opens a new file if it does not.

Switching Files

You can easily switch back and forth between files by using the [SHIFT]-[CTRL]-[~] command (press [~] while holding the [SHIFT] and [CTRL] keys down simultaneously). Using this method preserves the contents of all named buffers (but not the default buffer), search strings (used by **n** and **N** commands), etc., and the last operation (.) is also remembered. Thus if you add text to one file in a single operation, then switch files and move the cursor to a desired location in the other file and press the period (.) key, the same operation is repeated in the new current file.

The *ex* command, **:e #** (reopen previous file), when executed from *vi*, theoretically should behave the same way but it tends to be susceptible to requiring a **:w!** command before switching files, even when **autowrite** option is set, and, besides, the command requires more keypresses.

Intermediate Editing: File Manipulation Techniques

9

Vi users frequently need to perform file manipulations that include a variety of operations such as:

- Insert the contents of an existing text file into the file being edited.
- Copy part of the file being edited into another file.
- Copy all or part of the file being edited into one or more other files.
- Split the current file being edited into several smaller files.
- Edit two files simultaneously and use shared buffers to easily move text between the two files.

This list is not complete. There are many reasons and combinations of user needs that require writing all or part of the buffer file to other locations in the HP-UX file system.

In addition, you may need to access the HP-UX operating system to perform some necessary task without terminating the editing session. Perhaps you need to list a directory before selecting a filename to be used with a write command to store the current workfile. Or you may need to mail a file to someone or perform some other task. *Vi* provides a "shell-escape" capability so that you can exit to a user shell to perform the task, then return to *vi*.

These and other topics are discussed in this chapter.

Merging Another File into Text

Probably the most common file manipulation task is merging all of an existing file into text (*vi* cannot copy only part of a file; if you don't want the entire file, you must edit it first or copy it to a different file and edit the copy if the original must be preserved intact). The procedure for merging a file into the file being edited is simple. Simply give *vi* the following command:

`:r` *filename* RETURN

Vi then uses HP-UX to locate the file and copy it into the file being edited beginning **after** the current cursor line. Subsequent lines in the file being edited are pushed down to make room for the file being inserted. For example, consider the following excerpt from a sample file:

```
This line represents the early part of the file.
This line is the current cursor line.          ← Current cursor line
This line represents the remainder of the file.
```

The cursor is located anywhere on the second line when the read-file command is given. After the file is read and inserted, the cursor is moved to the first line of the inserted file (the line following the cursor line before the insertion began). Thus, the result looks like this:

```
This line represents the early part of the file.
This line is the current cursor line.
This line represents the inserted file.          ← New cursor line
This line represents the remainder of the file.
```

The cursor is relocated to the third line and positioned at the first visible character on the line.

Merging a File after a Text Pattern

You may occasionally need to merge a file into the file being edited starting after the line that contains a certain text pattern, although you may not know the location of the pattern in the file. Of course, you could search for the pattern then use the **:r** command to read the file being merged, but you can also combine an *ex* global search command with the read command to perform the same operation. Assuming you are already familiar with *ex* commands (described in Chapter 10), the command form is as follows:

:**g**/*text_pattern*/**r** *filename*

to read a file, or

:**g**/*text_pattern*/**r** !*HP-UX_command*

to read standard output from an HP-UX command or command sequence into the file. Also note that any standard *ex* line address form described in the chapter covering *ex* commands can be used instead of the **g** address form shown. For example, consider the following text segment from an earlier chapter in this manual:

```
However, be wary because if you do not exercise adequate care
in defining the string to be changed, you may get more changes
than you really wanted.  For example, changing every occurrence
of the to xyz in a file also changes Athena to
Axyzna; probably not what you would want.
:g/xyz/r junk_
```

The bottom (command line) on the display shows an *ex* command that is interpreted as follows:

g/xyz/	Tells *ex* to search every line in the file for the text pattern **xyz**.
r	Open a new line after any line containing the pattern **xyz** and fill it with the text contained in file *junk*. In this example, the file *junk* consists of a single line containing the text, "`junk file`"
underscore character	Represents the cursor). When RETURN is pressed, the following change occurs in the displayed text:

```
However, be wary because if you do not exercise adequate care
in defining the string to be changed, you may get more changes
than you really wanted.  For example, changing every occurrence
of the to xyz in a file also changes Athena to
junk file
Axyzna; probably not what you would want.
junk file
2 lines added
```

Note that since the pattern **xyz** appears on more than one line, the file is merged after each line where the text appears.

This technique can be used for various purposes such as form letters or other applications. The example shown should be sufficient for you to develop other more useful ideas.

Note

The example shown reads a file into the file being edited. However, changing the **r** (read file) to **w** (write file) does not write the line containing *text_pattern* to a file, but rather writes the entire file being edited to the specified file each time *text_pattern* is encountered, and is therefore not useful.

The Write Command: Saving All or Part of the Current Workfile

The **ZZ** command is most commonly used to save the current workfile at the end of an editing session and was discussed in an earlier chapter. However, there are times when you may want to save all or part of the current file in its present form elsewhere, then edit it further. One obvious method for doing so is to copy the file before invoking *vi* by using the HP-UX *cp* command. However, it is not uncommon to take an existing file and edit it before creating the copy. You can avoid having to terminate the edit before copying the file by using the **:w** command. As discussed in Chapter 2, the command:

:w *filename* RETURN

is used to save the current workfile in its present form in a specified file. You can also specify that only certain lines are to be copied into the file by specifying a starting and ending line. There are several ways to do this. For example,

```
:10,88w junk RETURN
```

tells *vi* to copy lines 10 through 88 into a file named *junk*. If *junk* already exists, you must use the forced form of the command:

```
:10,88w! junk RETURN
```

which overwrites (and destroys the previous contents of) the file *junk* if it already exists (if the file does not exist, a new file is created without complaint).

Other methods can also be used to identify beginning and last line. For example:

```
:.,.+2w junk RETURN
```

copies the current line and the two following lines into file *junk*. Note that the *ex* notation for line numbering (**.+2**) is used, rather than the *vi* cursor command sequence **2+**.

```
:.,$w junk RETURN
```

copies the current line through end-of-file into file *junk*.

When using this command, you can use any combination of addressing methods that is recognized by *ex* as described in Chapter 10. Several methods are illustrated in the remainder of this chapter.

Using File Markers

File markers can also be used to specify start and finish lines. Markers can be accessed by using the accent grave (`) or apostrophe (') in normal editor operation. The accent grave is used to move the cursor to the exact location where the marker was placed, while the apostrophe moves the cursor to the first visible character of the line where the marker was placed. When using file markers to locate beginning and ending line in a file write command, only the apostrophe form can be used. Thus, to write all of the current file from marker **a** through marker **c**, use the command:

```
:'a,'cw junk RETURN
```

If both markers reside on the same line of the file, only one line is written to the output file. If you attempt to write part of a line by placing both markers on the same line then using the accent grave form to identify cursor location, an error message results and no write operation is performed.

Using Text Patterns

As with any *ex* command that uses line addresses, a text pattern can be used instead of a line number to define beginning and/or ending line in the file being written. For example,

```
:../pattern/w! junk
```

writes all lines starting withe the current line through the line containing the text pattern *pattern* to the file *junk*. The exclamation point (!) forces the file to be overwritten if it already begins. Likewise,

```
:/pattern_a/,/pattern_b/w! junk
```

writes all lines starting with the line containing *pattern_a* through the line containing *pattern_b* to file *junk* with overwrite if the file exists.

Other combinations are also possible using addressing forms discussed in the chapters dealing with *ex* commands.

Appending to a File

You can also append to an existing file instead of writing to a file by using the "double redirection" symbol. For example, to write lines 1 through 25 to a file named *junk* then append lines 150 through 200 to the same file, use the sequences:

```
:1,25w junk RETURN
```

then

```
:150,200w >>junk RETURN
```

To append the entire workfile to an existing file named *existingfile*, use:

```
:w >>existingfile RETURN
```

If the named file does not exist, the specified text is written to a new file having the specified name. This is a useful way to write the file and ensure that any existing text is not overwritten as well as a convenient means of merging text from various sources.

Changing File Names

In the course of normal editing, you will commonly encounter the need to change the name of the file being edited before writing or closing a session, or storing various versions of a file under multiple names that are similar, yet unique.

Amending Current Filename During Write

Another version of the write command can be used to amend the name of the file slightly. For example, suppose you are currently editing file *filename* and execute the following command:

```
:w %1 RETURN
```

Vi remembers the filename used when it was invoked, in this case *filename*. The percent sign (%) is used to represent the name of the file currently associated with the buffer file. Thus, when the example command is executed by *vi*, the **1** is appended to *filename*, and the buffer is stored in file *filename1*. Note that if a directory pathname is associated with %, the new file will be stored in same directory as *filename*.

Other variations on these basic themes can be used. Possibilities should be self-evident as you gain experience.

Changing the Name of the Current File

Sometimes, as when creating various versions of a given file, it is helpful to be able to write the current file, change the current file name to something else, edit the file again and store it under the new name, and possibly continue *ad infinitum, ad nauseum.* To change the name of the current file, execute the *ex* **:file** command as follows:

```
:file new_filename RETURN
```

Piping the Workfile to a Command

All or part of the current workfile can be piped to the shell (actually a new shell is spawned) for filtering or other processing. A shell can also be spawned and results from the process inserted or appended to the current buffer file. The techniques used for performing such tasks are described in detail along with several examples in Chapter 11 which is entitled Advanced Editing: Shell Operations.

Escaping to an HP-UX Shell

Vi provides an escape mechanism for escaping to an external HP-UX shell where you can run programs, manipulate files, or perform other tasks. This is accomplished by entering External Mode (as discussed in Chapter 1) then using the shell-escape character (!) as follows:

:!*command* RETURN

Command can be any valid HP-UX command. If you need to perform several HP-UX operations before returning, you may prefer to spawn a new user shell from which you can execute the commands and later use a logout command or keypress sequence to return to the undisturbed editing session. You have several options.

You can use the *ex* command **sh** to spawn a new Bourne shell directly from the editor as follows:

:**sh** RETURN

This command can only be used to access the Bourne shell. It cannot be used to access any other shells such as C shell, Korn shell or a custom shell you might have written yourself.

A second method involves typing only one more character, but creates a different sequence of events to obtain an equivalent result. It also provides access to any available shell on your system. Here are three examples:

:!**sh** RETURN	to access the Bourne shell
:!**csh** RETURN	to access the C shell
:!**ksh** RETURN	to access the Korn shell

This command form behaves as follows:

- The **:!** sequence spawns a new user process (Bourne shell) to execute an HP-UX command.
- The remainder of the command is the name of a shell program name such as *sh*, *csh*, or *ksh*. The shell spawned by the **:!** sequence then spawns a new process to execute the shell you specified on the remainder of the line.

This means that the preferred method for spawning a Bourne shell is to use the *ex* command **:sh**, but for other shells, the **:!** form must be used. If you use the latter form to access the Bourne shell, you might get questioned by a System Administrator who has a preference for reducing the number of open processes on a multi-user system, but aside from that either approach is equally effective and does not measurably affect system performance.

When you are ready to return to the editor, type the normal logout sequence for the shell you are using or use the exit command if it is supported by the shell. This terminates the processes that were spawned during the detour from the editing session, and resumes the editing session. You will probably have to press another key such as RETURN or the space bar to redraw the screen if the **:!** command form was used. Also remember that you must eventually return to *vi* to prevent possible loss of the buffer file unless you have written it to permanent storage immediately before executing the shell-escape command.

Dealing with Special Characters

The command interpreter in *vi*/*ex* is similar in some respects to the C-shell interpreter in *csh*. Thus the special characters used by *csh* are also significant to *vi* when using shell escapes. This can be a problem if you are sending mail by using a *vi* shell escape. For example, suppose you used a command similar to the following to originate a mail message:

```
:!mailx ihxp5!netsys_a!corpsys_b!john
```

This tells *vi* to spawn a new shell to handle mail, run the mail program, and originate a message to *john* who resides on *corpsys_b* that can be accessed by our system by making two hops through backbone systems *ihxp5* and *netsys_a*. Or does it? *csh* uses the exclamation point (**!**) to represent the previous command. Other characters such as % are also significant to the command interpreter (% represents the current filename). To successfully mail a note, use a backslash (\) to escape the special character. Thus the previous command should be:

```
:!mailx ihxp5\!netsys_a\!corpsys_b\!john
```

If the address includes a path to an external mail handler such as HP-mail on HP 3000 systems where % is used, the % should also be preceded by a backslash.

csh special characters include the following:

```
! & | % + - * ? / ^ < > ( ) && || << >> # ;
```

and `$`

Which of these characters is interpreted as a special character depends on the context in which it is used. In any case, preceding the character with a backslash cancels its interpretation as a special character.

Using Tag Files to Edit Large or Multiple Programs

vi and *ex* include a tag-file capability that, when used in conjunction with the *ctags*(1) command (see *HP-UX Reference* for information about the *ctags* command), greatly simplifies editing random code segments in a large program or group of programs. This section uses a simple example to illustrate the general technique. Expanding the technique to extremely large multiple-file code structures, however, is not difficult.

The Program File

An example Amigo-protocol line printer driver program is shown in one of the appendices in the Device I/O Library tutorial (not contained in this volume). Here are two excerpts from that program which is about nine printed pages long in total. For this example, the program is stored in file *program_file* in the current directory. Each line in the file is preceded by a line number.

Excerpt from C Program Source File

Segment 1:

```
272  /* ROUTINE TO DO THE MAIN I/O TO THE BUS */
273  /* lock bus, do preamble, read/write, do postamble and unlock bus */
274  /* preamble must be 3 or 4 bytes, postamble must be 1 or 2 bytes */
275  int
276  HPIB_msg(rw_flag, pcm1, pcm2, pcm3, buffer, length, ocm0, ocm1)
277  int     rw_flag;
278  int     pcm1;
279  int     pcm2;
280  int     pcm3;
281  char    *buffer;
282  int     length;
283  int     ocm0;
284  int     ocm1;
285  {
286          unsigned char pre_cmd[4];
287          unsigned char post_cmd[2];
288          int tlog = -1;
289
290          pre_cmd[0] = UNL;       /* always issue unlisten command first */
291          pre_cmd[1] = pcm1;
292          pre_cmd[2] = pcm2;
293          pre_cmd[3] = pcm3;
294
295          post_cmd[0] = ocm0;
296          post_cmd[1] = ocm1;
```

```
297
298         /* first get exclusive use of the bus */
299         if (io_lock(eid) < 0)
300                 fatal_err("io_lock", ptr_dev, F_EXIT);
301
```

Segment 2:

```
375
376 /* do the amigo clear followed by selective device clear */
377 amigo_clear()
378 {
379         HPIB_msg(H_WRITE, MTA, DLA, SCG_BASE + 16, "\0", 1, SDC, UNL);
380 }
381
382 /* get the dsj byte */
383 int
384 amigo_dsj()
385 {
386         unsigned char dsj_byte[1];
387
388         HPIB_msg(H_READ, MLA, DTA, PR_SEC_DSJ, dsj_byte, 1, UNT, 0);
389         return(dsj_byte[0]);
390 }
391
392 /* return the amigo status byte */
393 int
394 amigo_status()
395 {
396         unsigned char status_byte[1];
397
398         HPIB_msg(H_READ, MLA, DTA, PR_SEC_RSTA, status_byte, 1, UNT, 0);
399         return(status_byte[0]);
400 }
401
402 /* output a buffer to printer */
403 amigo_write(buffer, length)
404 char *buffer;
405 int length;
406 {
407        int status, dsj = 0;
408
409        /* write the buffer */
410        HPIB_msg(H_WRITE, MTA, DLA, PR_SEC_DATA, buffer, length, UNL, 0);
411 again:
412         /* now wait for parallel poll response */
413         if (Debug)  printf("%s Ppoll wait\n", ptr_dev);
414         if (hpib_wait_on_ppoll(eid, 0x80>>devba, 0) < 0)
415                 fatal_err("hpib_wait_on_ppoll", ptr_dev, F_EXIT);
416
```

```
417         /* a DSJ is required to remove the ppoll response from device */
418         if (dsj = amigo_dsj()) {
419                 if (Debug) printf("%s DSJ = 0x%x\n", ptr_dev, dsj);
420
421                 status = amigo_status();
422                 if (Debug) printf("%s STATUS = 0x%x\n", ptr_dev, status);
423                 goto again;
424         }
425 }
426
427 /* output error message and conditionally abort */
428 fatal_err(message, fname, flag)
429 char *message;
430 char *fname;
431 {
432       fprintf(stderr, "%s: Error - %s of %s ", procnam, message, fname);
433       if   (errno) perror("");
434       else fprintf(stderr, "\n");
435
436       if (flag == F_RTRN) return;
437       if (flag == F_EXIT) exit(2);
438       exit(3);
439 }
```

Creating a Tags File

Before the tags option can be used with *vi*, the tags file must be created by the *ctags* command. To create a tags file on the example file, Execute the *ctags* command on file *program_file* as follows:

```
ctags program_file RETURN
```

This command produces a new file named *tags* in the current directory. File *tags* contains the following information:

```
   HPIB_msg        program_file    /^276  HPIB_msg(rw_flag, pcm1, pcm2, pcm3,
buffer, 1/
   Mprogram_file   program_file    /^128  main(argc, argv)$/
   amigo_clear     program_file    /^377  amigo_clear()$/
   amigo_dsj       program_file    /^384  amigo_dsj()$/
   amigo_identify  program_file    /^334  amigo_identify()$/
   amigo_set_pmask program_file    /^371  amigo_set_pmask()$/
   amigo_status    program_file    /^394  amigo_status()$/
   amigo_write     program_file    /^403  amigo_write(buffer, length)$/
   fatal_err       program_file    /^428  fatal_err(message, fname, flag)$/
```

Creating a Tags File for Multiple Files

ctags is most useful when you have a large collection of mixed C and Fortran (and possibly Pascal) source files. For example, suppose you have a collection of program files located under various directories that are, in turn, all collected under a single parent directory. For this example, suppose all C source file names end with **.c**, FORTRAN source file names end with **.f**, and Pascal source files end with **.p**. To create a single tags file, change to the parent directory of the directories containing the source files (*cd* command), then execute the command:

```
ctags */*.[cfp] RETURN
```

The result is a new tags file in the current directory. You can also specify other path names to the files of interest if they reside in other directory trees or subtrees.

When using this or an equivalent method, *ctags* complains if it creates a tag from a given file then finds another tag having the same name in another file. When such duplicate names are encountered, the first tag is retained and subsequent tags are not included. This is usually not a big problem, but care should be taken to avoid duplicate names. This condition can also occur when a program line looks like a procedure declaration but isn't.

Using the Tags File

Suppose you want to edit the code segment *amigo_identify*. Suppose further that file *program_file* is only one of a complex collection of programs and program segments and you don't have time to sift through 500 pages of source code listings to find which file contains the code segment. Fortunately for you, someone has run *ctags* on the entire set of files and the previous tags listing is only a small segment of the total tags file. You have three options:

- Use the **-t** option on the *vi* or *ex* command,
- Use the **:ta** command from *vi* or *ex* while editing a program source file, or
- While editing a program source file, place the cursor on the first character of the name of a program tag (such as on the line that calls a tagged subroutine) and press CTRL-].

Note

When using tag files, the current directory must be the directory containing the tags file. However, since the path to the file to be edited is specified on each line in the tag file, it is not necessary that the file being edited be in the same directory.

If starting a new session, execute the command:

```
vi -tamigo_identify RETURN
```

or:

```
ex -tamigo_identify RETURN
```

where the **−t** option tells *vi* (or *ex*) to use the specified tag from the tags file to determine which file to edit. *vi* searches the tags file for the identifier *amigo_identify* in the first column. The second column on the same line contains the name of the file where the code segment resides. *vi* opens the file, then uses the third column on the tag line as a search string to find the requested code segment. A few moments after the command is given, *vi* produces the following image on the screen with the cursor in the first column of line 334:

```
324             fatal_err("hpib_send_cmnd postamble", ptr_dev, F_EXIT);
325
326     /* at last unlock the bus so other bus users can access it */
327     if (io_unlock(eid) < 0)
328             fatal_err("io_unlock", ptr_dev, F_EXIT);
329
330     return(tlog);
331 }
332
333 int
334 amigo_identify()
335 {
336     unsigned char identify[2];
337
338     /* TLK31 (UNT) is special for amigo identify */
339     /* finish with a MTA (UNT is not save for non-amigo devices) */
340     HPIB_msg(H_READ, MLA, UNT, SCG_BASE + devba, identify, 2, MTA, 0);
341
342     switch(identify[0]) {
343     case 32:
344             /* Amigo identify */
345             switch(identify[1]) {
346             case 1:  return(T2608A);
"program_file" 439 lines, 14254 characters
```

Editing other Program Segments

Programmers often change several program segments during an editing session. This can become especially cumbersome as programs become large or involve a large number of files. Using tags to move around greatly simplifies the problem, and you can move from file to file without terminating *vi* (or *ex*) by using the **:ta** command.

Again using the previous example tags file, suppose you have modified *amigo_identify* and want to look at *HPIB_msg*. Simply execute the external-mode *vi*/*ex* command:

```
:ta HPIB_msg [RETURN]
```

or place the cursor under the `H` in `HP-IB_msg` on line 340 and press [CTRL]-[ ].

The editor examines the tag file then immediately moves back in the current file and displays line 276 at mid-screen with the cursor at the beginning of the line. Since the new tag is in the file currently being edited, no file change is needed so the editor does not write the buffer back into permanent storage before moving to the new tag.

When the New Tag Opens Another File

Suppose you decide to edit the file containing tag *new_tag*. Using the command:

```
:ta new_tag [RETURN]
```

(or [CTRL]-[ ]) causes one of two results. If the **autowrite** option has not been set (options are discussed in the chapter entitled Configuring the Vi/Ex Editor) and the file has not been preserved by using the **:w** command, the editor ignores the new tag command and displays the following error message at the bottom of the display screen:

```
No write since last change (:tag! overrides)
```

To recover, save the buffer by executing:

```
:w [RETURN]
```

or

```
:set autowrite
```

then repeat the **:ta new_tag** command.

If the **autowrite** option is set and the **:ta** command needs to open a new file, the current buffer is written back to permanent storage before the new file is opened.

Overriding Autowrite when Changing Files

You may occasionally need to change to another file after changing the current file, but for some reason you do not want to overwrite the original file (in other words, abort the edit and continue on another file). You can use the tag command to abort the current file without writing it back and open a new file by using the command form:

```
:ta! new_tag [RETURN]
```

The exclamation point after the **:ta** tells *vi* to abort (same as **:q!**) and open a new tagged file. To write the current file to a modified filename before changing can be done by using a write command where the filename is of the form *%text* where *text* is the character or characters used to alter the original filename as discussed earlier in this chapter under the topic Changing File Names.

Important

The tag command writes the current buffer **only** when changing files with the **autowrite** option set. The tag command followed by an exclamation point unconditionally aborts the current edited file and opens a new file. Any changes made to the current file are **always** lost when **:ta!** is used.

Using Ex Commands 10

This chapter discusses how to use *ex* commands from the *ex* editor and from *vi* as *ex*-mode commands (colon commands). Most commands are used identically from either editor. Any exceptions are explained in the command description.

Access to these commands is obtained from *vi* by typing a colon while in command mode. Start-up and use of *vi* is discussed in previous chapters starting with Chapter 2.

These commands can also be directly accessed when using *ex* instead of *vi*. Few HP-UX users use *ex* directly because most of them prefer to access *ex* features directly from *vi*. However, if you need to use *ex* directly, Chapter 13 describes the necessary procedures.

Colon Commands

The user prompt displayed by *ex* when it is ready for a new command is the colon. If you are in *vi* command mode, typing a colon switches *vi* to *ex* (external) mode so that you can use any appropriate *ex* command without leaving *vi* except for while executing the command. For this reason, *ex* commands are often referred to as colon commands when used from *vi*. To form a colon command, the colon character, whether an *ex* prompt or a *vi* command, is followed by a command sequence that can be any legitimate *ex* command. To execute the command, press [RETURN] or [ESC] after typing it.

Command Format

In general, *ex* commands follow a format resembling:

Standard form: :⟨*line_address*⟩⟨*command*⟩ ⟨*parameters*⟩⟨*count*⟩⟨*flags*⟩

or

Variant form: :⟨*line_address*⟩⟨*command*⟩! ⟨*parameters*⟩⟨*count*⟩⟨*flags*⟩

where all items except the colon are optional and can be included or omitted as the situation dictates. The function of each item when present in a command line is as follows:

Elements in a Command Line and Their Purpose

Element	Description
:	Start of command (*vi*) or prompt (*ex*). To avoid confusion over whether a colon was typed from *vi* or automatically provided by *ex* and whether another colon needs to be typed, the editor ignores the second colon if one is present (three colons, however, produce an error message).
⟨*line_address*⟩	Defines which line or lines are to be processed by the command. For commands that accept an address, zero, one, or two addresses may be present. If more addresses are provided than the command can use, extra addresses are ignored beginning with the first. If no address is present but one is needed by the command, the current line is assumed.
⟨*command*⟩	Defines the type of operation to be performed. If no command is provided, the last previous command is repeated.
⟨*command*⟩!	Specifies the variant form of ⟨*command*⟩ if available. Some default variants can be controlled by ⟨*options*⟩ in which case the ! serves to toggle the default.
⟨*parameters*⟩	Arguments or options to a command such as option names in a **:set** command, file name in an **:edit** command, regular expression in a **:substitute** command, or a target address for a **:copy** command.

⟨*count*⟩ Usually specifies the number of lines affected by ⟨*command*⟩. Optional or required, depending on the command. Value is rounded down if rounding is necessary.

⟨*flags*⟩ Identifies action to be taken upon completion of ⟨*command*⟩. Flag characters include **p** (print line), **c** (confirm each change before making it), and **g** (repeat ⟨*command*⟩ globally across the line). Any number of + or - characters can be used immediately before a flag to introduce an offset from the current line before executing the action specified by the flag. With the exception of the confirm and global flags associated with substitutions, flags are of little interest to *vi* users although they can be important when using *ex* on an electro-mechanical printer/terminal.

The topics that follow describe the components of a command line in greater detail.

Line Address Primitives

The first item after the colon is usually a line address of some form (if absent, the current line is assumed) unless the command does not require or does not accept an address. A missing address or a single address identifies a single line. Two line addresses separated by a comma specify the starting and ending line for a group of contiguous lines. *ex* recognizes the following line address forms which are sometimes referred to as *addressing primitives*:

Address	**Corresponding Line in the File Being Edited**
1	First line in file.
$	Last line in file.
.	Current line.
n	*n*th line in file.
.−*n*	*n*th line before current line.
.+*n*	*n*th line after current line.
%	Abbreviation for **1,$** which means every line in the file.

/⟨*pattern*⟩/ or ?⟨*pattern*⟩?	Searches forward (/) or backward (?) respectively for a line containing text that matches ⟨*pattern*⟩ where ⟨*pattern*⟩ is any regular expression, usually a string of text characters. The number of that line is then used as an address. Searches normally wrap around the end of the buffer file. If you only want to print the next line containing ⟨*pattern*⟩, the trailing / or ? can be omitted. If ⟨*pattern*⟩ is omitted or explicitly empty, the last previous regular expression used in a *pattern* search is substituted for ⟨*pattern*⟩.
'' or '⟨*marker_name*⟩	Used to locate previously-marked lines. Before each non-relative motion of the current line (.), the previous current line is marked with a tag, subsequently referred to by a double single quote (acute accent) character pair (''). Thus you can easily refer or return to this previous location. A line can also be marked by using the **mark** command followed by a ⟨*marker_name*⟩ consisting of any single lowercase letter in the range **a** through **z**. Marked lines can then be referred to in addresses by using ⟨*marker_name*⟩ preceded by a single quote (').

Combining Addressing Primitives for Multiple-Line Operations

One or more addressing primitives can be combined (with the use of appropriate separators) to specify a group of lines that are to be subjected to ⟨*command*⟩ where each separator character is a comma (,) or semicolon (;). Such address lists are evaluated from left-to-right. When addresses are separated by ; the current line address (.) is set to the value of the addressing primitive immediately preceding the ; before the next address is interpreted. If the command line contains more addresses than the command requires, all but the last one or two are ignored. If two addresses are required by the command the line identified by the first address must precede the second-line position in the buffer file. A zero-character address can be used in an address specification, in which case it defaults to the current line. Thus **,100** is equivalent to **.,100** and **,+5** is equivalent to **.,.+5**. Providing a prefix address for a command that expects none produces an error diagnostic.

Here are several examples of commonly used address forms including single-line addresses and combined addressing primitives for multiple-line operations:

Examples of Colon Command Line Address Forms

Address[1]	Affected Line(s)
1	First line in file.
n	Line *n* in file.
.	Current line in file.
.−4	Fourth line before current line in file.
.+8	Eighth line after current line in file.
$	Last line in file.
g	All lines in file.
1,.	All lines from beginning of file to current line.
.,$	All lines from current line to end of file.
.,.+5	Current line through fifth following line.
1,.+5	First line in file through fifth line after current line.
.−10,.+5	Tenth preceding line through fifth following line.

Pattern searches and markers can also be used to specify lines in the buffer file that are to be subjected to ⟨*command*⟩. Here are some examples of how they are used:

Address[1]	Affected Line(s)
/⟨*reg_exp1*⟩/	First line in forward search direction containing text pattern that matches the valid regular expression ⟨*reg_exp1*⟩.
?⟨*reg_exp1*⟩?	First line in backwards search direction containing text pattern that matches the valid regular expression ⟨*reg_exp1*⟩.
/⟨*reg_exp1*⟩/,/⟨*reg_exp2*⟩/	First line in forward search direction containing a text pattern that matches the valid regular expression ⟨*reg_exp1*⟩ through first following line in forward search direction containing a text pattern that matches the valid regular expression ⟨*reg_exp2*⟩. Second line must not precede first line in file due to end-of-file wrap-around.

[1] Space characters after the colon and on either side of the comma are optional but not normally used.

?⟨*reg_exp2*⟩?,?⟨*reg_exp1*⟩?	First line in backwards search direction containing a text pattern that matches the valid regular expression ⟨*reg_exp1*⟩ through first encountered line in same search direction containing a text pattern that matches the valid regular expression ⟨*reg_exp2*⟩. Line containing ⟨*reg_exp1*⟩ must not precede line containing ⟨*reg_exp2*⟩ in file due to beginning-of-file wrap-around.
``,.	All lines starting with previous current line through current line (previous line must precede current line in file).
'⟨*marker_1*⟩,'⟨*marker_2*⟩	All lines starting at line containing ⟨*marker_1*⟩ through the line containing ⟨*marker_2*⟩. ⟨*marker_2*⟩ must not precede ⟨*marker_2*⟩ in file. The name of each marker is any distinct lowercase letter in the range **a** through **z**.

Building the Command

Once the address has been defined and structured, you are ready to form the command part of the line. Most command names are English words, and the first letter in the word or a prefix form of the word is acceptable as an abbreviation. However, similarity between two commands can result in an ambiguous or conflicting abbreviations if not properly resolved. All such ambiguous abbreviations are resolved in favor of the more commonly used commands. Thus, for example, the abbreviated form of the **substitute** command is **s**, while the shortest available abbreviation for **set** is **se** because **substitute** is used more often by more people than **set**.

Command Parameters

The number and type of parameters associated with each command varies, depending on the command. This section describes those parameters, their purpose, and use.

Address Parameter

Many commands require one or two addresses where each address can have any of the valid forms previously described. Some commands can use either one or two addresses. If only one address is present, operations are performed relative to that line. If two addresses are present, they identify the starting and ending line in the text block that is to be processed by the command. If no address is provided on the command line but one is required by the command, the current line is assumed.

Command Name

The command name or its abbreviation comes after the address (if any). A complete list of command names and corresponding abbreviations follows this section along with a full description of each command and its use. Some commands require other information or parameters that are always appended following the command name; for example, option names in a *set* command, a file name in an *edit* command, a regular expression in a *substitute* command, or a target address for a *copy* command as in `1.5 copy 25` or `.,.+5 copy 12`.

Flags and Options After Commands

Various flags and options can be used after many commands to specify additional action to be taken upon completion of the command. They include the following:

Flag	Option	Description
	c	Confirm option used mainly with the **substitute** command. Confirm each change before continuing to next. Editor displays the line being changed with a circumflex (^) underneath each character that will be affected if the change is made. To accept the change, press [Y] then [RETURN]. To reject it, type [N] (or any other character except [Y] then [RETURN], or simply press [RETURN].[1]
	g	Global option. If proper conditions exist, execute ⟨*command*⟩ across entire line. This flag is most commonly used with the substitute command where a text pattern to be changed may exist more than once in a given line. If the **g** flag is not present, only the first matching pattern in the line is changed. With the **g** flag, the text pattern is processed every time it appears in the line.
p		Print flag. Print the current line after ⟨*command*⟩ has been processed. Has no effect if colon command is initiated directly from *vi*.
l		List flag. Print the current line after ⟨*command*⟩ has been processed, but also show the position of tab characters and end-of-line position. Has no effect if colon command is initiated directly from *vi*.
#		Print-line-number flag. Print the current line after ⟨*command*⟩ has been processed, but precede the printed line with its corresponding line number in the current buffer file.

vi users have no need to use the **P** flag since the display shows the current file contents at all times. The **r** option that is used with the **substitute** command is equivalent to the **&** command and is described later with the **substitute** command.

[1] This option does not work correctly on HP-UX Release 5.2 on Series 500 systems. It behaves as described, except that no changes are made in the buffer file, even if the change response is "yes".

If you are running *ex* directly (or after a **Q** command from *vi*) instead of accessing *ex* commands from *vi*, *ex* normally prints the new current line after each change, so *p* is still rarely necessary. Any number of + or - characters can also be given with these flags if you are using *ex* to move the current line in the specified direction. If any + or - flags are present in the command, the specified offset is applied to the current line value **before** the printing command is executed. For example, the command **:s/new/old/p+++** would print (display) the third line following the modified line.

Comments

Comment commands are ignored by the editor. This feature is useful when making complex editor scripts where explanatory comments are needed. Any line beginning with a double quotation mark (") is treated as a comment and no action results. Comments beginning with " can also be placed at the ends of commands, except in cases where they could be confused as part of text (as in shell escape sequences or in substitute or map commands).

Multiple Commands per Line

Multiple commands can be combined on a single line by separating adjacent commands with a | character. However, global commands, comments, and the shell escape ! must be the last command on a line because they are not terminated by a |.

Using the | Command Separator Character in :map Commands

The vertical bar character (|), since it is used as a command separator, must be escaped when using it in a **:map** command in order to protect it from interpretation as a separator between two commands on the same line. To escape the character, it must be preceded by a ^**V**. However, the ^**V** character is also interpreted as a special character, so it must, in turn be preceded by another ^**V**. Thus to place a vertical bar command separator character in an argument to the **:map** command, you must press CTRL+V twice, followed by the | key.

Reporting Large Changes

Most commands that change the editor buffer file contents give feedback whenever the scope of the change exceeds a threshold set by the report option. This feedback helps detect undesirably large changes so that they can be quickly and easily reversed with an *undo*. When using commands that have a more global effect (such as *global* or *vi*) you will be informed if the net change in the number of lines in the buffer file during this command exceeds the threshold.

Ex Command Descriptions

As we said before, all *ex* commands are constructed around a variation on the following form:

<address> <command> ! <parameters> <count> <flags>

where all parts are optional (or not allowed in some cases as described later). The simplest case in *ex* is the empty command which prints the next line in the file (if part of a *vi* colon command, it does nothing). To preserve user sanity when operating from *vi* or visual mode from *ex*, *ex* ignores a : preceding any command.

In the following command descriptions, the command, its standard abbreviation, and default value (if any) are shown in the left column.

Define an Abbreviation for Use as a Typing Aid (vi/ex)

Command Format	Command Description
:abbreviate ⟨*word*⟩ ⟨*text*⟩ or **:ab** ⟨*word*⟩ ⟨*text*⟩	Adds the specified abbreviation to the current list of abbreviations. ⟨*word*⟩ is the abbreviated form of ⟨*text*⟩ that is being defined by the command. When *vi* is in insert/append mode, if ⟨*word*⟩ is typed as a complete word (whitespace before and after), the editor expands the abbreviation, replacing it with ⟨*text*⟩. Defined abbreviations are discarded at end of editing session. No address is allowed with this command.

vi/ex maintains a table of abbreviations that can be used as typing aids. When a text pattern is typed that matches the abbreviation ⟨*word*⟩ and it is preceded and followed by a whitespace character, (space, tab, or end/beginning-of-line) the pattern is replaced by ⟨*text*⟩ in the buffer file and on the display screen. The table of abbreviations is destroyed at the end ofteach session. To construct a permanent set of abbreviations, use a *.exrc* file in your home directory as described in Chapter 12 entitled Configuring the Vi/Ex Editor.

For example, to abbreviate the term "cathode ray tube" as "crt" for ease in typing, use the command:

```
:abbreviate crt cathode ray tube RETURN
```
or
```
:ab crt cathode ray tube RETURN
```

Whenever you type the abbreviation, the editor will automatically expand it to the full form if it is preceded and followed by whitespace (space, tab, or end/beginning of line).

Cancel a Previously Defined Abbreviation (vi/ex)

Command Format	Command Description
:unabbreviate ⟨*word*⟩ or **:una** ⟨*word*⟩	Delete ⟨*word*⟩ from the list of abbreviations.

For example, to cancel the previous abbreviation for "crt", type the command:

```
unabbreviate crt RETURN
```
or
```
una crt RETURN
```

Append Text after Specified or Current Line (ex only)

Command Format	Command Description
:append ⟨*text*⟩ or **:a** ⟨*text*⟩ Default addr: current line Uses one address	Inputs one or more lines of new text, starting after the specified line. If the address preceding the append command is the current-line address (.) or absent, new ⟨*text*⟩ is placed after the current line. If address zero (0) is given, ⟨*text*⟩ is placed at the beginning of the buffer file. To terminate the append, type . at the beginning of a new line then press RETURN with no further text on that line. This command is not recognized by *vi* in *ex* mode.

This command is used to enter new text after the specified or current address. Press return after the **:a** or **:append**, type the text being added, then press RETURN after the last (or only) line being added followed by . and RETURN to terminate input.

For example, to add three lines after the 12th line following the current line, type:

```
:.+12a RETURN
This is added line 1. RETURN
This is added line 2. RETURN
This is added line 3. RETURN
. RETURN
```

Append Text but Toggle Autoindent Option (ex only)

Command Format	Command Description
:append! ⟨*text*⟩ or **:a!** ⟨*text*⟩ Default addr: current line Uses one address	Same as **append** except the variant flag on **append** toggles the setting of the autoindent option to its opposite state for the duration of text input. Upon termination of append, autoindent reverts to its normal state. This command is not recognized by *vi* in *ex* mode.

For more information about autoindenting, refer to the discussion near the end of Chapter 4 which treats the subject of manipulating text, and the **:set autoindent** command in Chapter 12, Configuring the Vi/Ex Editor.

Print Current Command Argument List (vi/ex)

Command Format	Command Description
:args or **ar**	This command prints the members of the argument list that was provided as part of the HP-UX command line when the editor was started. The current argument is delimited by left and right brackets in a form resembling: (...*arg arg* [*arg*] *arg arg* ...). No address or other arguments are allowed with this command.

This command is sometimes useful when you are editing several files in succession and need to know which file in the list of files is currently being edited or what other files are in the list.

This command can be used only in the following form, exactly as shown, with no other arguments:

```
:arg RETURN
```

Change One or More Lines to New Text (ex only)

Command Format	Command Description
:change ⟨*count*⟩ ⟨*text*⟩ or **:c** ⟨*count*⟩ ⟨*text*⟩ Default addr: current line Uses one or two addresses	Replaces lines specified by ⟨*count*⟩ with the input ⟨*text*⟩. Upon completion, the current line becomes the last line in ⟨*text*⟩. If no text is provided, the command is treated as a **delete**. Uses one address if ⟨*count*⟩ is specified or implied or two address if changing multiple lines without using ⟨*count*⟩.

This command can be a bit confusing because it can take several line addressing forms including:

First and Last Address Specified:

:⟨*start_addr*⟩,⟨*end_addr*⟩**c** RETURN

Starting Line and Number of Changed Lines Specified:

:⟨*start_addr*⟩**c**⟨*line_count*⟩ RETURN

Start at Current Line with Number of Changed Lines Specified:

:c⟨*line_count*⟩ RETURN

Replacement text is then typed on subsequent lines following the form described for the **append** command.

The templates above show the general form of the command. Any accepted address can be used, whether line number, marker, or regular expression; provided it can correctly identify the line or lines being affected by the command.

Here is an example where 13 lines are replaced by three lines starting at the second line after the current line:

```
:.+2c13 RETURN
This is changed line 1. RETURN
This is changed line 2. RETURN
This is changed line 3. RETURN
. RETURN
```

Change One or More Lines to New Text but Toggle Autoindent (ex only)

Command Format	Command Description
:change! ⟨*count*⟩ ⟨*text*⟩ or **:c!** ⟨*count*⟩ ⟨*text*⟩ Addressing: same as above	Same as **:change** except the variant flag on **:change** toggles the setting of the autoindent option to its opposite state for the duration of text input.

Change Current Directory (vi/ex)

Command Format	Command Description
:cd ⟨*directory_name*⟩	Changes the current working directory to the directory specified by ⟨*directory_name*⟩. If **autowrite** is set and the file has been modified, the buffer file is written to permanent storage before the change is made. If **autowrite** is not set, the file must be written to permanent storage before the command can be executed. To change directories without writing the current buffer file to permanent storage, use the command form **:cd!** ⟨*directory_name*⟩.

This command has no effect unless the **:edit** ⟨*file*⟩ or **:n** ⟨*file*⟩ command is used. The current buffer file is written to permanent storage (if autowrite is set) or discarded as appropriate, and the new file is opened from the new directory specified by the **:cd** command. If autowrite is not set, the current buffer must be written before the directory can be changed unless you override with a **:cd!** command. This lockout feature prevents you from changing directories then writing the current buffer into the wrong directory after making the change. You can use this feature to move from directory to directory and edit various files without terminating and restarting *vi* or *ex* between each file. When you terminate the editor program with **ZZ** or the **quit** command, the current directory changes back to what it was before editing started. This command does not affect the location of the directory where the buffer file is stored (the buffer directory is determined by the **:set directory** command described in Chapter 12).

Any legitimate directory and pathname descriptors can be used when specifying ⟨*directory_name*⟩, including .. and . If the **:cd!** form of the command is used, the buffer file is not lost. If this command is used during an editing session, commands such as **:w** and **ZZ** write the buffer to a file whose name matches the current filename being edited. Thus, if you are editing a file named *file_a* in directory *directory_X* then change to *directory_Y*, **:w** or **ZZ** writes the buffer to a file named *file_a* in *directory_Y*, creating a new file if necessary. If *directory_Y* already contains a file of that name, the existing file is overwritten by the new buffer file contents.

Copy One or More Lines to New Location (ex only)

Command Format	Command Description
:copy ⟨*address*⟩ ⟨*flags*⟩ **:co** ⟨*address*⟩ ⟨*flags*⟩ or **:t** ⟨*address*⟩ ⟨*flags*⟩ Default addr: current line	A copy of the lines specified by the address primitives that precede the **copy** command is placed after the line identified by ⟨*address*⟩ which can be zero. Upon completion, the current line (.) addresses the last line of the copy. The **:t** command is a synonym for **:copy**. Addressing rules are the same as for the **change** command with or without ⟨*count*⟩. ⟨*flags*⟩ is pertinent only when using *ex* instead of *vi*.

The addressing parameters preceding the **copy** command are similar to the addresses preceding the **change** command. A single address (or implied single address) copies one line to the specified target address ⟨*address*⟩. Two addresses, when present or implied, specify the starting and ending line when multiple lines are to be copied.

Commands can have any of the following general formats:

First and Last Address Specified:

:⟨*start_addr*⟩,⟨*end_addr*⟩**co** ⟨*destination_address*⟩ RETURN

Starting Line and Number of Copied Lines Specified:

:⟨*start_addr*⟩**co**⟨*line_count*⟩ ⟨*destination_address*⟩ RETURN

Start at Current Line with Number of Copied Lines Specified:

:co⟨*line_count*⟩ ⟨*destination_address*⟩ RETURN

Here are some examples:

- Copy the second line after the current line, placing it after Line 13 in the current file:

  `:.+2co13` RETURN

- Copy the current line through the second line after the current line (three lines total), placing them after Line 22 in the current file:

  `:.,.+2co22` RETURN

- Copy the entire buffer file, placing it after the last line. This produces the equivalent of two files concatenated head-to-tail in a new single file and is equivalent to the HP-UX command, `cat buffer_file >>buffer_file` :

  `:%co$` RETURN

Flags:
The **copy** command accepts the following flags:

#	Print current line preceded by line number after copy.
p	Print current line without line number after copy.

Encrypt Files (vi/ex)

Command Format	Command Description
:cr or :X	If your system is properly licensed and has file encryption software installed, this command provides editing for encrypted files.

Delete One or More Lines (vi/ex)

Command Format	Command Description
:delete ⟨*buffer*⟩ ⟨*count*⟩ ⟨*flags*⟩ or **:d** ⟨*buffer*⟩ ⟨*count*⟩ ⟨*flags*⟩ Default addr: current line	Removes the lines specified by ⟨*count*⟩ and ⟨*flags*⟩ from the text buffer file, and the line following the last line deleted becomes the new current line. If the deleted lines were originally at the end of the text buffer file, the new last line in the file becomes the current line. If a named ⟨*buffer*⟩ is specified by a single letter, the deleted lines are saved in that buffer. If the buffer name is lowercase, previous buffer contents are overwritten; if uppercase, the lines are appended to any existing text in the buffer.

Again, the addressing parameters are similar to the **copy** and **change** commands. A single address deletes a single line unless the delete command is followed by a ⟨*count*⟩ parameter. A pair of addresses identifies a block of lines to be deleted provided no ⟨*count*⟩ parameter is present. If ⟨*count*⟩ is present in the command line, the second address is used and ⟨*count*⟩ lines are deleted starting at that address (the first address is ignored). To store the deleted lines in a named buffer, specify the buffer name between the **delete** command and the ⟨*count*⟩ parameter as indicated above.

If a **p**, **l**, or **#** flag is included in the command and *ex* is being used instead of *vi*, the new current line after the deletion is completed is printed as specified by which flag is present.

First and Last Address Specified:

:⟨*start_addr*⟩,⟨*end_addr*⟩**d** RETURN

Starting Line and Number of Deleted Lines Specified:

:⟨*start_addr*⟩**d**⟨*line_count*⟩ RETURN

Current Line and Number of Deleted Lines Specified:

:**d**⟨*line_count*⟩ RETURN

Current Line, Number of Deleted Lines, and Overwrite Buffer "u" Specified:

:**d u** ⟨*line_count*⟩ RETURN

Current Line, Number of Deleted Lines, and Append to Buffer "u" Specified:

:**d U** ⟨*line_count*⟩ RETURN

Examples

Here are some more examples that are similar to those given for **change** and **copy** but showing some variations on the use of ⟨*pattern*⟩ addressing:

- Delete 3 lines starting at second line after the first encountered line containing *text*, and append to buffer **r**:

```
:/text/+2,dR3 RETURN
```

- Delete the current line and the next two lines after the current line:

```
:.,.+2d RETURN
```

- Delete all lines, starting at the 5th line after the line that contains marker **a** and continuing through the 4th line before the last line in the file:

```
:'a+5,$-4d RETURN
```

Edit a Different File (vi/ex)

Command Format	Command Description
:edit ⟨*file*⟩ **:e** ⟨*file*⟩ or **:ex** ⟨*file*⟩	Terminates the current editing session and starts a new session on the specified ⟨*file*⟩. If the **autowrite** option is not set (see Chapter 12) and the current file has been modified but not written, the command is aborted and an error message is displayed. If **autowrite** is set and the file has been modified, the current buffer is written to permanent storage before the new file is loaded into the buffer for editing. This command is commonly used from *vi* as well as *ex*.

That the three names and abbreviations for this command are identical to three HP-UX commands used to access various personalities of the *ex* editor is purely coincidental. When used as commands within the editor, **edit**, **e**, and **ex** are synonymous with each other. They are also synonymous with the synonymous HP-UX commands *e* (if it exists on the system in the form of a link to the *vi/ex* editor program file) and *ex*.

When this command is given, the editor checks the current file to see if it has been modified. If the file has been modified, the editor checks the **autowrite** option to see if it is set. If **autowrite** is not set, an error message is sent to the terminal and the command is aborted. If **autowrite** is set, the buffer file is written to permanent storage and the buffer is destroyed.

When the current buffer is cleared (current file not modified or modified file copied to permanent storage), the editor then examines the specified new file to make sure that it is a valid text file. If so, it is copied to the buffer area for editing and a new session begins in the usual manner.

After ensuring that this file looks reasonable, the editor reads the source file into its buffer. If the transfer is completed without error, the number of lines and characters in the file is displayed. Any 8-bit characters in the file are handled as 8-bit values in order to maintain compatibility with the international language capabilities that are supported on Series 800 HP-UX systems and Series 300/500 systems starting at Release 5.2.

Under certain conditions, the new file is sometimes treated as a modified file. If the last line in the new file is missing its trailing newline character, one will be supplied and a complaint message will be displayed. The current line (.) is the last line in the new buffer file.

The **:e** command and its variants cannot be used with a multiple filename list, unlike the **:n** (next) command.

Edit a Different File; Forced Command Version (vi/ex)

Command Format	Command Description
:edit! ⟨*file*⟩ **:e!** ⟨*file*⟩ or **:ex!** ⟨*file*⟩	The variant form of the **:edit** command terminates the current editing session, destroys the current buffer file whether it has been modified or not, then copies the specified new ⟨*file*⟩ to the editor buffer for editing. Any error or complaint messages that would normally result when switching to a new file without saving the current file are suppressed. Even if the **autowrite** option is set, the buffer is not written to permanent storage before it is destroyed.

Edit New File Starting at Specified Address (vi/ex)

Command Format	Command Description
:edit+⟨*n*⟩ ⟨*file*⟩ **:e+**⟨*n*⟩ ⟨*file*⟩ **:edit+/**⟨*pattern*⟩ **:e+/**⟨*pattern*⟩	Same as **edit** command, but causes the editor to begin at line ⟨*n*⟩ rather than at the last line or start with the first line containing ⟨*pattern*. **Important:** ⟨*pattern*⟩ must contain no spaces or tabs.

Print Current File Name and Description (vi/ex)

Command Format	Command Description
:file or **:f**	Prints the current file name and provides the following information: whether the file has been modified since the last write command; whether it is read-only; current line number; number of lines in the buffer; and the relative location of the current line in the buffer (expressed as a percentage). This command is equivalent to the *vi* command CTRL-G.

Change Name of Current File (vi/ex)

Command Format	Command Description
:file ⟨*file*⟩	The current filename is changed to ⟨*file*⟩ which is considered **not modified.**

Process all Lines Containing ⟨pattern⟩ (vi/ex)

Command Format	Command Description
:global/⟨*pattern*⟩**/**⟨*commands*⟩ or **:g/**⟨*pattern*⟩**/**⟨*commands*⟩ Default: All lines in file	Scans each line among those specified and marks those lines that match the regular expression in ⟨*pattern*⟩. Current line is then set to each marked line in sequence and ⟨*commands*⟩ are then executed from each successive new current line before advancing to the next marked line.

The ⟨*commands*⟩ parameter represents all of the remaining commands on the current input line. The current input line can consist of multiple lines, provided that each line except the last line in the multiple-line command ends with a slash (/) which is interpreted as a line-continuation flag. If ⟨*commands*⟩ (and possibly the trailing (/) delimiter after ⟨*pattern*⟩) is omitted, each line matching ⟨*pattern*⟩ is printed. This means that these two commands are equivalent:

```
:g/⟨pattern⟩/p
:g/⟨pattern⟩
```

Append, insert, and change commands and their associated new input ⟨*text*⟩ are permitted. The (.) terminator normally required for insert, append, and change can be omitted if the end of the ⟨*text*⟩ associated with the command coincides with the end of the **global** command line or lines. **:open** and **:visual** commands can also be used in the command list, but their input text must be typed from the terminal keyboard because the commands do not accept text as arguments. When accessing the **global** command from *vi*, ⟨*commands*⟩ must not contain any commands that cannot be used from *vi* (such as the **append** command).

The **:global** command itself must not appear in ⟨*commands*⟩. The **:undo** command is also not permitted in ⟨*commands*⟩ because **:undo** could have the effect of reversing the entire global command. The **autoprint** and **autoindent** options are inhibited during global command. The / delimiter at the end of input lines may also be inhibited. Th value of the report option is temporarily set to an infinite value (disabled) so that sc that reporting can be based on the entire global operation. The context mark (") is set to the current line (.) address before the global command begins. It remains unchanged during the global command with the possible exception of an **:open** or **:visual** within the global.

Process all Lines that Do Not Contain ⟨pattern⟩ (vi/ex)

Command Format	Command Description
:global!/⟨*pattern*⟩**/**⟨*commands*⟩ **:g!/**⟨*pattern*⟩**/**⟨*commands*⟩ or **:v/**⟨*pattern*⟩**/**⟨*commands*⟩	The variant form of **global** runs ⟨*commands*⟩ on each line that does not contain text that matches ⟨*pattern*⟩. Note that the **v** command is synonymous with **g!**, but there is no **v!** variant.

Insert New Text Before Specified or Current Line (ex only)

Command Format	Command Description
:insert ⟨*text*⟩ **:i** ⟨*text*⟩ Default addr: current line Uses one address	Places ⟨*text*⟩ before the specified line. Upon completion, the new current line becomes the last line in ⟨*text*⟩. If no ⟨*text*⟩ is given, it is set to the line before the addressed line. This command is equivalent to append except that the new text is inserted in the file before the current line instead of after it. As with **append**, ⟨Text⟩ must be terminated by a line that contains a period in the first column position with no other characters on the line (except for certain situations in **:global**). This command is not recognized by *vi* as a valid colon command.

Insert New Text but Toggle Autoindent (ex only)

Command Format	Command Description
:insert! ⟨*text*⟩ **:i!** ⟨*text*⟩ Default addr: current line Uses one address	Same as **:insert** except the variant flag on **insert** toggles the setting of the autoindent option to its opposite state for the duration of text input. Upon termination of insert, autoindent reverts to its normal state. This command is not recognized by *vi* as a valid colon command.

For more information about **insert!**, refer to the earlier text related to the **append!** command.

Join (combine) Lines on Single Line and Trim Whitespace (vi/ex)

Command Format	Command Description
:join ⟨*count*⟩ ⟨*flags*⟩ **:j** ⟨*count*⟩ ⟨*flags*⟩ Default addr: current and next line Uses one or two addresses	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 period, question mark, or exclamation point at the end of the line, or none if the first following character is a right-hand parenthesis. If there is already white space at the end of the line, the white space at the start of the next line is discarded.

Addressing

If ⟨*address*⟩ and ⟨*count*⟩ are absent, current and next line are combined. If two addresses are present without a ⟨*count*⟩, the lines starting with the first address and ending with the last address are joined into a single line. If ⟨*count*⟩ is specified, only one address is used (if two addresses are present, the first is ignored) and ⟨*count*⟩ lines are printed. If ⟨*count*⟩ is specified but no address is given, ⟨*count*⟩ lines are joined, starting with the current line.

Combine Multiple Lines on Single Line with Retained Whitespace (vi/ex)

Command Format	Command Description
:join! ⟨*count*⟩ ⟨*flags*⟩ **:j!** ⟨*count*⟩ ⟨*flags*⟩	The variant of **:join** causes a simpler join with no white-space processing. Adjacent lines are simply concatenated, and no whitespace is eliminated. Addressing is same as **join** command.

Print Text Showing Tabs and End-of-Line (vi/ex)

Command Format	Command Description
:list ⟨*count*⟩ ⟨*flags*⟩ or **:l** ⟨*count*⟩ ⟨*flags*⟩ Default addr: current line Uses one or two addresses	Prints the specified lines so that tab characters and end-of-line are recognizable. Tabs are printed as CTRL-uppercase **I** (**^I**), and the end of each line is marked with a trailing **$**. The last line printed becomes the new current line.

Map Text Pattern or Macro to a Function Key (vi only)

Command Format	Command Description
:map ⟨*key*⟩ ⟨*replacement*⟩	This command can be executed from *vi* or *ex*, but the macro or text pattern produced by the definition can only be used when in *vi* mode. **map** is used to define macros that are associated with specific keyboard key codes. This form of the command has an effect only when *vi* is in command mode. ⟨Key⟩ should be a single character, or the sequence "#n" where ⟨*n*⟩ is a digit referring to function key ⟨*n*⟩. When the character or key specified by ⟨*key*⟩ is typed from **vi**, the corresponding ⟨*replacement*⟩ expression is substituted (and displayed if appropriate). On terminals that do not have function keys, you can type **#n** to represent the missing key. Examples of how this command is used are shown in greater detail in the example *exrc* file examples in Chapter 12 entitled Configuring the Vi/Ex Editor.

Command Format	Command Description
:unmap ⟨*key*⟩ ⟨*replacement*⟩ or **:unm** ⟨*key*⟩ ⟨*replacement*⟩	The macro expansion defined by a previous **:map** command for ⟨*key*⟩ is cancelled.

Command Format	Command Description
:map! ⟨*key*⟩ ⟨*replacement*⟩	Same as **:map** command but variant form has effect in both command and insert/append mode.
:unmap! ⟨*key*⟩ ⟨*replacement*⟩ or **:unmap!** ⟨*key*⟩ ⟨*replacement*⟩	The macro expansion defined by a previous **:map!** command for ⟨*key*⟩ is cancelled.

Mark Current or Specified Line (vi/ex)

Command Format	Command Description
:mark ⟨*x*⟩ **:ma** ⟨*x*⟩ or **:k** ⟨*x*⟩ Default addr: current line Uses only one address	Assigns the specified marker name ⟨*x*⟩ to the current line. ⟨*x*⟩ is a single lowercase letter that must be preceded by a blank or a tab. The marker name ⟨*x*⟩ can then be used as an address in subsequent commands to specify this line. The current line does not change. **:mark**, **:ma**, and **:k** are synonymous. A blank or tab between **:k** and the marker name is optional.

Move One or More Lines to a New Location (vi/ex)

Command Format	Command Description
:move ⟨*address*⟩ or **:m** ⟨*address*⟩ Default: Current line only Uses one or two addresses	Deletes the specified lines and copies them to a new location immediately after the line identified by ⟨*address*⟩. The first of the moved lines becomes the new current line. This command is functionally equivalent to a **copy** followed by a **delete** or a **delete** followed by a **put**. Refer to the **copy** command for information about line addressing for the lines being moved.

Edit Next File in Argument List (vi/ex)

Command Format	Command Description
:next or **:n**	Starts editing next ⟨*file*⟩ in argument list. If the file currently being edited has been saved or has not been modified since the session began, the new file is opened. If the current file has been modified but has not been saved, it is written to permanent storage if autowrite is set and opens the new file. If autowrite is not set and the file has been modified, an error message is produced and the command to open the next file is aborted. Used to open next file when HP-UX *vi* or *ex* command includes multiple filename arguments and you are ready to open the nex file. Previous file in argument list must be written unless **autowrite** option is set.

More About ⟨file⟩

Within the text of ⟨*file*⟩, the characters % and # are expanded as the current working file name and the alternate file name respectively (as when working with two alternate files and switching between them).

Force Edit Next File in Argument List (vi/ex)

Command Format	Command Description
:next! or **:n!**	This variant of the **next** command forces the termination of the current edit file without rewriting to permanent storage whether the file has been modified or not. Any warnings or error messages that would normally result from terminating the session and starting a new file are suppressed. Any changes that may have been made to the current file are irretrievably discarded.

Print Line(s) Preceded by Corresponding Buffer Line Number (vi/ex)

Command Format	Command Description
:**number** ⟨*count*⟩ ⟨*flags*⟩ :**nu** ⟨*count*⟩ ⟨*flags*⟩ :**#** ⟨*count*⟩ ⟨*flags*⟩ Default: current line	Prints one or more lines with each line preceded by its line number in the buffer file. Can be used with one address (specified or implied) and optional ⟨*count*⟩ or two addresses with no ⟨*count*⟩ specified as described below. The last line printed becomes the new current line.

Addressing

Addressing is the same as for the **print** command. If ⟨*address*⟩ and ⟨*count*⟩ are absent, current line is printed. If two addresses are present without a ⟨*count*⟩, all lines starting with the first address and ending with the last address are printed. If ⟨*count*⟩ is specified, only one address is used (if two addresses are present, the first is ignored) and ⟨*count*⟩ lines are printed. If ⟨*count*⟩ is specified but no address is given, ⟨*count*⟩ lines are printed, starting with the current line.

Here are some example command structures. Addresses can take any form previously described in the section entitled Line Address Primitives. ⟨*count*⟩ can be any numerical value that does not exceed the boundaries of the buffer file. Each command line shows a different form of the **number** command:

First and Last Address Specified:

:⟨*start_addr*⟩,⟨*end_addr*⟩**number** RETURN

Starting Line and Number of Changed Lines Specified:

:⟨*start_addr*⟩**nu**⟨*line_count*⟩ RETURN

Start at Current Line with Number of Changed Lines Specified:

:#⟨*line_count*⟩ RETURN

Enter Open Mode (ex only)

Command Format	Command Description
:open ⟨*flags*⟩ or **:open/**⟨*pattern*⟩**/**⟨*flags*⟩ Default: current line Uses only one address	Enables open mode editing on the current or addressed line. Open mode operation essentially amounts to changing to *vi* instead of *ex* except that only the current line is printed as with normal *ex* operation instead of displaying a full-screen file window as with normal *vi*. Thus you can use all of the normal *vi* commands such as **i**, **I**, **a**, **A**, **o**, **O**, **cw**, **cW**, **dw**, **dW**, **cf***x*, and so forth. If a ⟨*pattern*⟩ is used to define the line address, the cursor is initially placed at the beginning of the string matched by the pattern. [ESC] is used at the end of each change just as in *vi*. Cursor control keys such as **h**, **j**, **k**, and **l** as well as the *n***G** and other commands can be used to move from one line to another. To exit **open** mode, use **Q**. For more information about *vi* editor commands and operations, consult Chapters 2-4. The **Q** command used to exit **open** mode is related to, but not the same command as the **Q** command used to switch from *vi* to *ex*.

The **open** command is of little interest to most users who have ready access to display terminals, but can be useful for bringing many *vi* features to printing terminals for more interactive operation than can be obtained from *ex* in normal line mode or if you are using *ex* to circumvent *vi* on an unknown terminal type that is not supported by the *terminfo* database.

Emergency File Preservation (vi/ex)

Command Format	Command Description
:preserve or **pre**	You may, on rare occasion, encounter an error during a **write** operation, placing the contents of the buffer file in jeopardy if you follow it by a **quit** command. Use the **preserve** command to hold the buffer just as it would be in a system crash. This command is for emergency use 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 from your System Administrator.

Print One or More Lines (vi/ex)

Command Format	Command Description
:print ⟨*count*⟩ **:p** ⟨*count*⟩ **:P** ⟨*count*⟩ Default: print only current line	Prints one or more lines with non-printing characters displayed as control characters in the format "^x" (DEL is printed as ^?). Can be used with one address (specified or implied) and optional ⟨*count*⟩ or two addresses with no ⟨*count*⟩ specified as described below. The last line printed becomes the new current line.

Addressing

Addressing is the same as for the **number** command. If ⟨*address*⟩ and ⟨*count*⟩ are absent, current line is printed. If two addresses are present without a ⟨*count*⟩, all lines starting with the first address and ending with the last address are printed. If ⟨*count*⟩ is specified, only one address is used (if two addresses are present, the first is ignored) and ⟨*count*⟩ lines are printed. If ⟨*count*⟩ is specified but no address is given, ⟨*count*⟩ lines are printed, starting with the current line.

Here are some example command structures. Addresses can take any form previously described in the section entitled Line Address Primitives. ⟨*count*⟩ can be any numerical value that does not exceed the boundaries of the buffer file. Each command line shows a different form of the **print** command:

First and Last Address Specified:

:⟨*start_addr*⟩,⟨*end_addr*⟩**print** RETURN

Starting Line and Number of Changed Lines Specified:

:⟨*start_addr*⟩**p**⟨*line_count*⟩ RETURN

Start at Current Line with Number of Changed Lines Specified:

:P⟨*line_count*⟩ RETURN

Put Yanked or Deleted Text back in File (vi/ex)

Command Format	Command Description
:put ⟨*buffer*⟩ or **:pu** ⟨*buffer*⟩ Default: current line	Restores previously deleted or yanked lines after the line specified in the **put** command. Normally used with delete to move lines, or with yank to duplicate lines. If no buffer is specified, the last deleted or yanked text is restored. By using a named buffer, text can be restored that was saved there at any previous time in the current session.

Abort Editing Session but Protect Buffer (vi/ex)

Command Format	Command Description
:quit or **:q**	Terminates the editing session if the current buffer is not modified or has been saved if modified. If the buffer file has not been modified, termination is immediate. If the buffer file has been modified and the autowrite option is set (see Chapter 12 for information about setting options), the buffer is written to permanent storage before terminating the session. If autowrite is not set, an error message is displayed indicating that the file has been modified but not written and the command is aborted. To save the changes, use a **write** command then a **quit** or combine both commands using the form **:wq**. To abort the session without saving the changes, use the **:q!** command variant to force termination.

Abort Editing Session and Discard Buffer File (vi/ex)

Command Format	Command Description
:quit! or **:q!**	Forced **quit** terminates the editing session without saving the buffer file regardless of whether or not **autowrite** option is set. Unconditionally discards any changes to the current buffer. This command is commonly used to force termination of the editor when a buffer file has been badly damaged by inappropriate edits and when a list of files was specified for editing but you do not want to edit the remaining files in the list.

Merge File from File System into Buffer File (vi/ex)

Command Format	Command Description
:read ⟨*file*⟩ or **:r** Default addr: current line Uses only one address	Copies the entire contents of text file ⟨*file*⟩ into the editing buffer beginning after the current line if no address is specifed or after the specified line if an address primitive is provided before the **read** command. Specifying address zero inserts the file before the first line in the buffer. Upon completion of the read, the number of lines and bytes read are displayed as when starting a new session.

Filename Specification

In most situations, this command is very simple and does exactly what is expected: it reads a specified file into the current file after the current or specified line; especially when accessed from *vi*. However, under certain circumstances the result may be different:

- **No Current Filename:** If no filename argument is provided when the HP-UX command is given to start the session, the editor opens a buffer file, but does not assign a filename to it for permanent storage. If this condition exists when the **read** command is executed, the filename specified with the **read** command is assigned to the buffer. This means that when you execute a **write** command to save the buffer in permanent storage, the file you read when the **read** command was executed will be overwritten by the **write** command. This may not be what you want to happen (play it safe – specify a working file name when you start the session or be very careful about specifying a file name every time you use **write** during the session).
- **No Filename Specified for read Command:** If you use the **read** command but do not specify a filename, the current file being edited is used instead. This means that the file as it existed at the start of the session (or after the last write in the current session, if any) is inserted into the buffer file which again may or may not be what you want.
- **No Filename Specified and Buffer is Empty:** If the current buffer is empty (you are editing a new file or you deleted all lines from the existing file) and the read command has no filename specified, an error message is displayed.

Merge Standard Output into Buffer File (vi/ex)

Command Format	Command Description
:read !⟨*hpux_command*⟩ or **:r !**⟨*hpux_command*⟩ Default addr: current line	Reads standard output from any HP-UX command ⟨*hpux_command*⟩ in the editor buffer starting on the next line after the line secified by ⟨*address*⟩. This is not a variant form of **read**, but rather a **read** command that inputs standard output from an HP-UX command sequence directly into the buffer instead of sending it to the terminal display. A blank or tab before the (!) is strongly recommended, and may be mandatory in some earlier versions of *vi/ex*.

More About ⟨hpux_command⟩

Within the text of ⟨*command*⟩, the characters **%** and **#** are expanded as the current working file name and the alternate file name respectively (as when working with two alternate files and switching between them).

Recover File After Hangup, Power Fail, or System Crash (vi/ex)

Command Format	Command Description
:recover ⟨*filename*⟩ or **:rec** ⟨*filename*⟩	Recovers ⟨*filename*⟩ from the system save area. Used after an accidental hangup of the phone, a system crash, or after a **preserve** command. You will be notified by mail when a file is saved (except after a **:preserve** or a modem hangup). You can also recover the preserved file by using the **−r** option to the *vi* or *ex* command at the beginning of the session as described near the end of Chapter 2 which discusses recovery procedures.

Rewind Argument List to First Argument (vi/ex)

Command Format	Command Description
:rewind or **:rew**	This command, which apparently got its name from its similarity to rewinding tapes, rewinds the argument list associated with the HP-UX command that started the current editing session. The file corresponding to the first file in the list is then copied to the buffer and opened for editing. The same restrictions apply with respect to the autowrite option and modified files as for any new file when changing from the current file to a new file. See the description of the **next** command earlier in this section for details.

Rewind Argument List to First Argument and Discard Current Buffer (vi/ex)

Command Format	Command Description
:rewind! or **:rew!**	Rewinds the argument list, discarding any changes made to the current buffer.

Set or List Editor Options (vi/ex)

Command Format	Command Description
:set ⟨*parameter*⟩ **:set** ⟨*parameter*⟩**?** **:set all** **:set** No address allowed Abbreviated form: **:se**	Sets or lists current editor configuration parameters. If ⟨*parameter*⟩ is included after the command, that parameter is set to the value specified. If ⟨*parameter*⟩ is followed by a question mark (⟨*parameter*⟩**?**) the current setting of that parameter is printed or displayed. The **set all** command lists all available parameters with their current settings, while **set** lists only those options whose values have been changed from their defaults. Chapter 12 entitled Configuring the Vi/Ex Editor provides a detailed discussion of this command.

Create New Shell from Editor (vi/ex)

Command Format	Command Description
:shell or **:sh**	Creates a new shell. When you terminate the shell with a CTRL-D, the shell dies and editing resumes at the same location in the file. This is a convenient way to temporarily depart from the editor, change directories if you wish, and perform other tasks or take care of any other spur-of-the-moment need then return without terminating the edit or disturbing the editing environment such as current directory. A typical use is when you are editing a program and need to use the *man* command to look at a manual page entry so you can verify an option or some other operating detail then return.

Spawning New Shells

This command spawns a new shell as specified by the **:set shell** configuration command (default is the Bourne shell). An alternate form is used by some operators:

:!sh RETURN

However, when this form is used, the exclamation point after the colon spawns a shell which, in turn, spawns the Bourne shell (a :!csh command, on the other hand spawns a C shell). If you use this method, do not be surprised if your System Administrator asks you why you need so many shells operating at the same time (the practice is really rather harmless but you can avoid the new shell process by using the command **:!exec sh**, **:!exec csh**, or **:!exec ksh**).

On the other hand, if you prefer to use the C shell (or some other shell if it is available on your system) use the less elegant form to specify the alternate shell. For example, to get a C shell, use:

:!csh RETURN

This command spawns a Bourne shell (! shell escape) which, in turn executes the HP-UX command *csh*, thus spawning a new C shell.

Return Conditions

To return to the editor, use a normal shell termination command. For example, to terminate a Bourne or C shell, press CTRL-D or type:

```
exit RETURN
```

The spawned shell dies, the intermediate shell, if one was spawned, also dies, and control returns to the editor program.

Input Ex Editor Commands from a File (vi/ex)

Command Format	Command Description
:source ⟨*file*⟩ or **:so** ⟨*file*⟩	Causes the editor to read and execute *ex* commands from the specified commands file ⟨*file*⟩. Commands in the file can be nested.

This command provides a means for collecting a series of *ex* commands into a file then using that file to edit another file. If you are editing a large number of files, the streaming editor *sed* would probably be a better choice, but this command provides a convenient means for collecting several global changes that need to be made on a file before you manually make other changes that are not suitable for execution from a commands file.

Note

On Series 200, 300, and 500 systems up to and including HP-UX Release 5.2 and Series 800 Release 1.0 and 1.1 this command does not work correctly when accessed from *vi* in *ex*-command mode due to a bug in the *vi* program. From *vi*, **:so** executes the first command in the file then returns without executing the remainder of the file. However, the command works correctly when using the *ex* editor. To work around the problem when using *vi*, use the **Q** command in *vi* command mode to change to *ex*, then type the **so** command after *ex* displays its colon prompt. When *ex* provides another prompt after completing the command file, type **vi** RETURN to resume editing with *vi*.

Substitute Text Within Line or Lines (vi/ex)

Command Format	Command Description
:substitute/⟨*pattern*⟩**/**⟨*repl*⟩**/** ⟨*options*⟩ ⟨*count*⟩ ⟨*flags*⟩ or **:s/**⟨*pattern*⟩**/**⟨*repl*⟩**/**⟨*options*⟩ ⟨*count*⟩ ⟨*flags*⟩ Default addr: current line Uses one or two addresses	On each line as defined by the one or two addresses preceding the **substitute** command or the combination of an address and ⟨*count*⟩, the first text encountered that matches the regular expression ⟨*pattern*⟩ is replaced by the replacement text pattern defined by ⟨*repl*⟩. If the global (**g**) option is present in the command line, all occurrences are substituted. If the **confirm** option (**c**) is specified, you are asked to confirm each substitution beforehand. The line to be substituted is displayed (with the string to be substituted marked with '^' characters underneath). To accept the substitution, type **y**. Any other input causes no change to take place. After a **substitute** the current line is the last line substituted.

Options and Flags

Option or Flag	Description of Action Taken
c	Confirm each change before making it.
g	Perform the change for every occurrence of ⟨*pattern*⟩ in each addressed line.
r	This option is used without specifying a search pattern or replacement text. The **:**⟨*address*⟩**s r** command reuses the search pattern and replacement text from the last **substitute** command, but on the lines specified by the current ⟨*address*⟩, and is equivalent to the **:**⟨*address*⟩**&** command in the same situation.
p	Print current line after copy without line number.
#	Print current line preceded by line number after copy.
l	Print current line after copy and show tabs and end-of-line position.

Lines can be split (only when in *ex*, not from *vi* colon command) by substituting newline characters into the line. The newline in ⟨*repl*⟩ must be escaped by preceding it with a backslash (\). Other metacharacters available in ⟨*pattern*⟩ and ⟨*repl*⟩ are described below.

Repeat Most Recent Substitution

Command Format	Command Description
:substitute ⟨**options**⟩ ⟨*count*⟩ ⟨*flags*⟩ or **:s** ⟨**options**⟩ ⟨*count*⟩ ⟨*flags*⟩ Addressing: same as above	If ⟨*pattern*⟩ and ⟨*repl*⟩ are omitted from the **substitute** command, the last previous substitution is repeated. **substitute** without ⟨*pattern*⟩ and ⟨*repl*⟩ is equivalent to the **&** command.

Using Tags to Edit a New Location

Command Format	Command Description
:tag ⟨*tag*⟩ or **:tag** ⟨*tag*⟩	Changes the defined current line from its present locaton to a new location defined by ⟨*tag*⟩ which may be in the current file or in a different file. If ⟨*tag*⟩ is located in a different file, the present file is saved in permanent storage and the file containing the new ⟨*tag*⟩ is copied to the buffer and opened for editing. Refer to Chapter 9 entitled File Manipulation Techniques for more information about creating and using tag files.

A file named *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 in each line contains the name of the tag, the second field is the name of the file where the tag resides, and the third field contains an addressing primitive that can be used by the editor to find the tag in the file specified in the second field. The address field is usually a contextual scan using /⟨*pattern*⟩/ to maintain immunity from minor changes in the file. Scans for /⟨*pattern*⟩/ are always performed with the **nomagic** option temporarily set for the duration of the scan, independent of its normal setting.

⟨*tag*⟩ names in the *tags* file must be sorted alphabetically. Sorting is done automatically by the *ctags* command (see *ctags*(1) in the *HP-UX Reference* for more information about *ctags*).

Reverse (Undo) Changes Made Previously

Command Format	Command Description
:undo or **:u**	Restores all changes made in the buffer by the most recent buffer editing command to their original form prior to the command. If the command included **global** operations, all changes resulting from the global command are treated as a single operation, whether the changes are made by *vi*, *ex*, or *ex* in **open** mode. However, commands that interact with the file system such as **write** and **edit** cannot be undone. If you use **undo** to reverse a change only to find that the result is not what you wanted, the changes can be restored by another **undo** before executing any other command that alters the buffer file. In other words, **undo** is its own inverse.

Current Line

Undo always assigns the file mark ' to the current line prior to performing any **undo** changes so that you can readily return to that position by pressing the backwards single quote twice (''). When the **undo** is completed, the new current line is usually the first line restored or the line before the first line deleted if no lines were restored. But for commands such as **global** that affect larger blocks of text, the current line position is not changed by the **undo**.

Print Editor Version Number and Last Change Date

Command Format	Command Description
:version or **:ve**	While this command is rarely of interest to most users, it may be useful on occasion if you need to identify what version of the *vi*/*ex* editor you are using. The **version** command prints the current version number of the editor and the date the editor program was last changed.

Change from Ex to Vi Editor or from Vi to Ex

Command Format	Command Description
:visual ⟨*type*⟩ ⟨*count*⟩ ⟨*flags*⟩ or **:vi** ⟨*type*⟩ ⟨*count*⟩ ⟨*flags*⟩ Default addr: current line	Changes from *ex* to **visual** operation and displays a text window whose height and location in the file is defined by the parameters provided on the command line. Uses only one address (specified or implied), whether or not ⟨*count*⟩ is specified.
Q	While in *vi* command mode, the **Q** command switches the editor to its *ex* personality.

Window Size and Location

If no parameters are provided with the **vi** command, a full-screen window or default window size defined by the baud rate between the terminal and the HP-UX computer is created with the current line located at the top of the window. If ⟨*type*⟩ is specified, window size, text placement, and format are as follows:

- ⟨***type***⟩ **not specified:** A window ⟨*count*⟩ lines high and ending at the bottom of the display screen is displayed with the current or addressed line at the top of the window. If ⟨*count*⟩ is not specified or if it is greater than the available screen size, a full screen display window is used.
- ⟨***type***⟩ **= −:** Same as ⟨*type*⟩ not specified except that the current or addressed line is placed at the bottom of the window.
- ⟨***type***⟩ **= .:** Same as ⟨*type*⟩ not specified except that the current or addressed line is placed at the center of the window.
- ⟨***type***⟩ **= #:** Displays a full window starting at the current or addressed line but each line is preceded by its corresponding line number in the editor buffer. ⟨*count*⟩ cannot be used with this ⟨*type*⟩.

Returning to Ex from Vi

To return to *ex* from *vi* or to change from *vi* to *ex* at any time, press **Q** (SHIFT-Q) while in *vi* command mode. **:Q** cannot be used from *vi* and causes an error message if attempted.

Edit Another File with Vi

Command Format	Command Description
visual ⟨*file*⟩ or **visual +n** ⟨*file*⟩ Abbreviated forms: **vi** ⟨*file*⟩ or **vi +n** ⟨*file*⟩	These forms of the *ex* **visual** command are equivalent to comparable forms of the *ex* **edit** command except that the visual editor is used instead of the line editor. See the **edit** command for more information about each form.

Write All or Part of Buffer to a Permanent File

Command Format	Command Description
:write ⟨*file*⟩ or **:w** ⟨*file*⟩ Default addr: entire file Uses one or two addresses	Writes the contents of the current buffer to the specified ⟨*file*⟩. If ⟨*file*⟩ is not specified, the buffer is written to the original file currently being edited. If two addresses are specified, all lines starting at the first address through the line identified by the second address are written to the specified or default file. If only one address is provided, only one line is written. If no address is included in the command line, the entire file is written.

The most common form for this command is simply

```
:w RETURN
```

which writes the current buffer back to the original file if it exists. If no file exists (new file edit), a file is created then written to. If no filename was specified at the beginning of the edit, a new file is created and the current file name is changed to ⟨*file*⟩ before writing the new file.

If ⟨*file*⟩ already exists, and the filename does not match the current file, an error message is displayed and the write operation is aborted. This protects files from being accidentally overwritten in a moment of carelessness or inattention. If the current file name matches the specified ⟨*file*⟩, the file is overwritten even if you don't want it overwritten (unless the **readonly** flag is set).

The current line is not changed by this command. If an error occurs while writing the file, the editor treats the buffer file as a modified file whether it has been modified or not since the beginning of the session.

Append All or Part of Buffer to a Permanent File

Command Format	Command Description
:write >>⟨*file*⟩ or **:w >>**⟨*file*⟩ Default: entire file	Equivalent to the previous **write** command description except that the buffer or specified lines are appended to ⟨*file*⟩ which must already exist (an error message is displayed if it does not exist).

Force Write All or Part of Buffer to a Permanent File

Command Format	Command Description
:write! ⟨*name*⟩ or **:w!** ⟨*name*⟩ Default addr: entire file Uses one or two addresses	This variant form of the **write** command overrides checking for an existing file and forces a write to the named file if the file system permissions allow the write operation to proceed. Note that there is no space between the write command and the variant flag (**!**). If the file does not already exist, this command is equivalent to **:write**.

Write All or Part of Buffer to an HP-UX Command

Command Format	Command Description
:write !⟨*command*⟩ or **:w !**⟨*command*⟩ Default: entire file Uses one or two addresses	This is not a variant form of the **write** command, but rather a shell-escape form of the **write** command. The Writes the specified lines into command. Note the difference between **:w!** which overrides checks and **:w !** (space before **!**) which writes to a command.

More About ⟨command⟩

Within the text of ⟨*command*⟩, the characters **%** and **#** are expanded as the current working file name and the alternate file name respectively (as when working with two alternate files and switching between them).

Write then Quit: Terminate a Session

Command Format	Command Description
:wq ⟨*name*⟩	Combined write file and terminate session command on a single line. Similar in effect to the *vi* command **ZZ**.
:wq! ⟨*name*⟩	Variant form of **write then quit** does not protect an existing file from being overwritten (equivalent to **:w!**) and terminates the session as soon as the write operation is complete.

Terminate Editing Session

Command Format	Command Description
:xit ⟨*name*⟩ or **:x** ⟨*name*⟩	Equivalent to the *vi* command **ZZ**. Terminates the editing session. If the buffer file has been modified but not yet written to permanent storage, the file is saved before terminating the editor. If the file is not writable (readonly option set or no write permission on file being edited, the exit command is aborted so you can save the buffer in a different file before it is discarded.

Yank Text into a Buffer for Use in Copy Operations

Command Format	Command Description
:yank ⟨*buffer*⟩ ⟨*count*⟩ ⟨*flags*⟩ or **:y** ⟨*buffer*⟩ ⟨*count*⟩ ⟨*flags*⟩ Default addr: current line	Copies the lines specified by ⟨*count*⟩ and ⟨*flags*⟩ from the text buffer file into the specified ⟨*buffer*⟩ or into the default buffer if no buffer name is provided. The current line does not change. ⟨*buffer*⟩ can be any lowercase letter in the range **a** through **z**. If a buffer name in the range **A** through **Z** is specified, yanked text is appended to the named buffer instead of replacing it.

Note

If the *ex* commands **yank** and *put* are used directly from *vi* as colon commands to copy text from one location to another, **undo** may not work correctly on Series 200, 300, 500 HP-UX systems through Release 5.5 and Series 800 through Release 1.1. Use the *vi* commands for yank and put (which work correctly) instead as explained in earlier chapters of this tutorial.

Addresses

yank can be used with one or two addresses specified. One address with no ⟨*count*⟩ specified copies the specified line into the named buffer. Two addresses with no ⟨*count*⟩ specified copies multiple lines starting and ending with the specified lines. Two addresses and a ⟨*count*⟩ value copies ⟨*count*⟩ lines starting at the second specified address (the first address is ignored). If no address is specified, the current line address is used by **yank**.

Examples

First and Last Address Specified:

:⟨*start_addr*⟩,⟨*end_addr*⟩**ya** RETURN

Starting Line and Number of Yanked Lines Specified:

:⟨*start_addr*⟩**ya**⟨*line_count*⟩ RETURN

Current Line and Number of Yanked Lines Specified:

:**ya**⟨*line_count*⟩ RETURN

Current Line, Number of Yanked Lines, and Buffer *e* Specified:

:**ya e** ⟨*line_count*⟩ RETURN

Print Window Containing ⟨count⟩ Lines

⟨type⟩ not Specified

Command Format	Command Description
:z ⟨*count*⟩ Default addr: next line	Print ⟨*count*⟩ lines starting at next line after addressed line. If address is omitted, printing starts at next line after current line. If ⟨*count*⟩ is not specified, a default window is printed which contains the number of lines specified by the **scroll** option to the **set** command, provided it does not exceed display screen capacity (see Chapter 12 for details). Upon completion of the command, the current line is changed to the last line printed in the window.

⟨type⟩ Specified

Command Format	Command Description
:z ⟨*type*⟩ ⟨*count*⟩ Default addr: current line	Similar to the preceding **:z** command, except a ⟨*type*⟩ parameter specifies the placement of the addressed line in the printed text window. If ⟨*count*⟩ is not specified, the default window contains twice the number of lines specified by the **scroll** option to the **set** command described in Chapter 12, provided the number of lines does not exceed display screen capacity. Upon completion of the command, the current line is changed to the last line printed in the window.

Text Positioning in the Window

⟨*type*⟩ determines the position of the displayed text in the window as follows:

⟨type⟩	description
− (minus)	Addressed line (or current line if no address primitive is provided) is placed at the bottom of the window.
. (period)	Places the addressed (or current) line at the center of the window.
+	Displays window of lines following specified line. Successive **z+** commands scroll down through the buffer.
^	Displays window of lines that are two windows prior to the specified line. Successive **z^** commands scroll up through the buffer.
=	Displays specified line at center of window with a line of 40 dashes (−) above and below the specified line. The current line becomes the specified line (not the last line of the window as with the other ⟨*types*⟩).

No other values for ⟨*type*⟩ are accepted. ⟨*count*⟩ specifies the window length in lines, if present. If no window length is given, the current value of the **scroll** option to the **set** command (see Chapter 12) is doubled and used as a window length, provided the value does not exceed available screen display space. If any window size value exceeds screen capacity, the full screen is used. If a full-screen window is needed, the display screen is cleared before displaying the new window.

Execute a Shell Command

Command Format	Command Description
:! ⟨*command*⟩	The shell-escape character (!) in this command is an editor command that spawns a new shell and sends the remainder of the command following the exclamation point to the shell for execution. When the command has been executed by the shell, the spawned shell dies and control returns to the editor. When this command is executed from *ex*, a single ! is printed on a line by itself after the text currently displayed on the terminal screen. The prompt for the next command is printed on the following line. If this command is executed from *vi*, the message: `[Hit return to continue]` is displayed instead (press any key to continue with *vi*). If the buffer has been modified since the beginning of the session or since the last buffer **write** command to permanent storage, whichever occurred later, the warning message **No write since last change** is displayed. However, if the shell escape command is followed by another shell escape command before any other changes are made in the file, no warning is given on the second escape.

More About ⟨command⟩

Within the text of ⟨*command*⟩, the characters % and # are expanded as the current working file name and the alternate file name respectively (as when working with two alternate files and switching between them). The character !, if present in ⟨*command*⟩, is replaced with the text of the last previous command. Thus, in particular, !! repeats the last such shell escape. If any such expansion is performed, the expanded line is echoed to the terminal screen. The current line position in the buffer is not changed by this command.

Repeat Previous Shell-Escape Command

Command Format	Command Description
:!!	Repeats the most recent shell-escape command. The existing shell command buffer is again sent as command input to a newly spawned shell.

Pipe Part or All of Buffer to a Command (vi/ex)

Command Format	Command Description
:(⟨*addr1*⟩,⟨*addr2*⟩)!⟨*hpux_command*⟩ Uses one or two addresses Default: none Address must be specified	Copies all text lines within the specified line address range to HP-UX standard input for processing by ⟨*hpux_command*⟩. Standard output from ⟨*hpux_command*⟩ is then returned to the editor where it replaces the original lines that were sent to standard input. If no address is specified and the command requires input from standard input, *stdin* is switched to the keyboard and the command hangs, waiting for keyboard input.

Print Current or Addressed Line Number

Command Format	Command Description
:= Default addr: last line in file Uses only one address	Prints line number of current or addressed line. If an address primitive is provided before the = command, the line number of the addressed line in the buffer file is printed. If no address precedes the command, the line number of the last line in the file (address $) is printed instead. The location of the current line remains unchanged.

Shift Lines Left or Right

Command Format	Command Description
(.,.)⟩ ⟨*count*⟩ ⟨*flags*⟩ (.,.)⟨ ⟨**count**⟩ ⟨*flags*⟩	Perform intelligent shifting on the specified lines; ⟨ shifts left and ⟩ shifts right. The quantity of shift is determined by the shiftwidth option and the repetition of the specification character. Only white space (blanks and tabs) is shifted; no non-white characters are discarded in a left-shift. The current line becomes the last line which changed due to the shifting.

Execute a Buffer (vi/ex)

Command Format	Command Description
: * ⟨*buffer_name*⟩	Execute the contents of buffer ⟨*buffer_name*⟩ as a valid *ex* command. The buffer must contain a valid *ex* command that does **not** begin with a colon. This feature is useful for yanking a complex editing command that has been placed in the file being edited, executing it, and being able to edit the command then yank and execute it again to fix errors in the command or perform a similar but different operation. It the command is accessed while in *vi*, a colon must precede the asterisk.
@ ⟨*buffer_name*⟩	This alternate form executes a valid *ex* command stored in ⟨*buffer_name*⟩ directly from *vi*. The command residing in the buffer must begin with a colon. The **@ command cannot be used from the ex** editor (use the * command instead).

Miscellaneous Commands

Command Format	Command Description
:^D	An end-of-file from a terminal input scrolls through the file. The scroll option specifies the size of the scroll, normally a half screen of text.
:⟨*address_1*⟩,⟨*address_2*⟩ or :⟨*address_1*⟩,⟨*address_2*⟩ |	An address or pair of addresses without a command causes the addressed lines to be printed. A blank line (no address or command) prints the next line in the file.
:⟨*address*⟩**&** ⟨*options*⟩ ⟨*count*⟩ ⟨*flags*⟩ 0, 1, or 2 addresses	The ampersand command repeats the previous **substitute** command.
:⟨*address*⟩~ ⟨*options*⟩ ⟨*count*⟩ ⟨*flags*⟩ 0, 1, or 2 addresses	Equivalent to the **&** command if the most recent previous regular expression was the search pattern part of a **substitute** command. If the most recent regular expression was part of a **global** command (not a global flag on a substitute command), you can use a tilde (between slash characters) as a regular expression and supply new replacement text to save retyping the previous expression.

The ~ command can be used, for example, to search for a pattern using a **global** command, review the lines that contain patterns matching the regular expression, then construct a new substitute command with appropriate address or addresses, then complete the command with the form:

s/~/⟨*new_text*⟩

and add any flags or options that may be appropriate. The **&** command can then be used to repeat the command on a new set of addresses, if desired.

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 the the previous regular expression used elsewhere (referred to as a previous scanning regular expression). The previous regular expression can always be referred to by a null *re*, e.g. '//' or '??'.

Magic and nomagic

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

The remainder of this discussion of regular expressions assumes that the setting of this option is *magic*.

Using Ex Commands

Editing the Command

If you make an error while typing a colon command, use [BACK SPACE] to move the cursor left to the appropriate position, then retype the rest of the command. As with normal *vi* operation, characters are not erased from the screen as you move the cursor left, but they are removed from the *vi*/*ex* command buffer. Hence, any extra characters that are not obliterated by retyping are ignored (you will notice that they disappear from the bottom line of the display as soon as you press [RETURN] or [ESC]).

An alternate and sometimes easier method for correcting a command is pressing the terminal "kill" charcter (usually [CTRL]-[U]) which immediately moves the cursor to the first character following the colon so you can retype the entire line as a different command.

If the command is complicated and you want to be able to easily change or fix errors in it, type the command as a line in the file, yank the line to a buffer, and execute the buffer. Procedures for doing so are described in Chapters 5 and 6.

Aborting or Changing the Command

If you type part of a colon command then decide you want to do something else instead, you can abort the command by using [BACK SPACE] or [CTRL]-[U] followed by [BACK SPACE] to back the cursor up to the left margin past the colon. When the cursor passes the colon, the the editor abandons the command and returns the cursor to where it was prior to the aborted colon command.

An easier method is to simply press the [BREAK] key ([DEL] produces the same result). This method has the side-effect of setting the HP-UX *vi* command return status flag to FALSE when *vi* terminates, but unless you are operating in an unusual environment, using the BREAK key should present no discernable disadvantage.

Aborting After Execution Begins

You may discover, particularly when performing global operations on a very large file, that you gave an incorrect command (such as inadvertently pressing / or ? instead of :) or an inappropriate command, and need to abort it. Press [BREAK]. Command execution stops, and an error message is displayed:

```
Interrupt
```

The cursor may or may not return to its original position prior to the command, and the file may or may not be untouched by the command, depending on what was happening at time of interrupt. If changes were made before the command was interrupted, you can use the **u** (undo) command to repair the damage and return to the pre-command state. After the undo, the cursor may move to a different location in the file, depending on the situation.

Undoing Colon Commands

Like normal *vi* commands, the external-mode commands are also subject to the **u** command. If you discover that the change you made did not produce the desired effect, press **u** immediately before executing any other command. As usual, if any command is executed after the colon command, the undo option for that command is forever lost, and you must either use another command or set of commands to fix the error or abort the session (**:q!** command) and start over.

Global Searches

Suppose you are working on a large file such as a large computer program or text file and need to look at every line in the file that contains a certain word, program label reference, or operand name. Rather than using a cumbersome series of / or ? followed by **n** or **N** search sequences, you can print all occurrences of the desired text pattern with a simple command of the form:

:**g**/*text_pattern*/**p** [RETURN]

where *text_pattern* is any regular expression of the form described in the tutorial on regular expressions earlier in this volume that is compatible with *vi* and *ex*. The **g** command specifies that the search is to be made globally (on every line) throughout the file, and the **p** command specifies that the results are to be printed on the display screen. Experienced users will recognize that this command is very similar to the HP-UX *grep* command.

After the lines are printed to the screen, the message:

```
[Hit return to continue]
```

appears at the bottom of the screen. Press any typing key to restore the normal editor display.

Limited Searches

You can easily limit the search for a given expression to a certain part of the file by specifying the starting and ending line numbers. Here is the command form:

:*start_line*,*end_line*g/*text_pattern*/p RETURN

where *start_line* is any valid line number identifier that specifies the starting line and *end_line* specifies the the last line in the search space. Valid line specifiers can be the actual line number (**1** is the first line in the file, **$** specifies the last line), line locations relative to current line, or any other form recognized by *ex*.

Finding Tabs and other Control Characters

Suppose you need to determine whether and where any control characters might be hidden in a file. This can be particularly important when examining a computer program file as well as in many other circumstances.

The **l** command accomplishes this task quite handily with the form:

:*start_line*,*end_line*l RETURN

Any control characters contained within the specified file segment are displayed in "hat" format, "hat" being a common vernacular name among UNIX users for the circumflex character (^). Tabs are displayed as **^I**, and end-of-line is displayed as **$**. For example, consider the following rather innocent looking line of text:

```
If this looks like a simple sentence, look between the words
```

Placing the cursor anywhere on the line and executing the command:

:.l RETURN

reveals more than what meets the eye:

```
If this looks like a^Isimple sentence, look between the words  ^I  $
```

showing two hidden tabs plus several spaces at the end of the line. Likewise, a command of the form:

```
:.,.+101 RETURN
```

displays all control characters in the current plus the 10 following lines.

After listing the lines, press any key to restore the normal editor display.

Advanced Editing: Shell Operations 11

As you gain experience, you will commonly encounter situations where you need to directly access HP-UX commands and capabilities from within the *vi* editor without terminating or disturbing the editing session. *vi* command mode supports several methods for accessing HP-UX commands, opening a vast field of possibilities for the imaginative user.

When accessing HP-UX commands, all or part of the current *vi* buffer is used as the standard input source for thoes HP-UX commands requiring standard input, and as standard output when the command standard output is to be placed in the text file. This ability to use the buffer as the source for standard input as well as the destination for standard output opens a wide realm of possibilities that fall into three general categories:

All or part of the current buffer can be sent to a series of one or more HP-UX commands whose final output is brought back to replace the original text that was sent to standard input. The *tee* command can also be used to send the output of any of the commands in the series to another file. This technique for accessing HP-UX commands is commonly called piping the buffer to a command.

In addition, all or part of the buffer can be sent as standard input to an HP-UX command in a write operation where the command output is sent elsewhere and is not brought back to replace the original text. This is called writing the buffer to a command. A comparable read operation imports standard output from an HP-UX command, and is called reading from a command.

This chapter presents several typical examples of each of these types of operations, discusses some useful techniques, and is intended to serve as a source of ideas that you can use in your own text editing applications.

For more information about the HP-UX commands used in the examples in this chapter, refer to the corresponding manual page entry in Section 1 of the *HP-UX Reference.*

Operation Types

As mentioned, shell operations fall into three general categories:

- Operations that modify (replace after processing) existing text in the file,
- Operations that insert new text from an HP-UX command into the file after the current line, and
- Operations that send existing text in the file to an HP-UX command and produce output elsewhere without altering the existing buffer contents.

The remainder of this chapter addresses each category separately.

Text Replacement Shell Operations

When shell operations are used to modify text currently residing in the *vi* buffer, that portion of the buffer being modified by HP-UX is sent by *vi* as standard input to the specified HP-UX command or commands. Standard output from the command is then sent back to *vi* which uses it to replace the original text. The text from *vi* that was sent to HP-UX can also be used or stored elsewhere by using the *tee* command in addition to being brought back for replacement in its modified form.

When you use this method for altering text, the command to *vi* that specifies the work to be done causes *vi* to spawn a new shell in response to the shell escape sequence. The spawned shell process then executes the shell command specified in the remainder of the shell escape command line that spawned the process. The command can consist of a single shell program/command, or it can be a series of commands piped together. Upon completion of the specified shell command, standard output from the last command is returned to the editor and replaces the original text object. The editor **undo** command can be used to restore the original text if the result is not what was wanted.

Text Replacement: Command Format

Text replacing shell operation commands generally have the form:

⟨**!**⟩⟨*count*⟩⟨*text_object*⟩⟨*shell_command*(s)⟩ RETURN

Element	Description
⟨**!**⟩	Exclamation point (**!**) is used to escape from *vi* to the shell interpreter.
⟨*count*⟩	Number of ⟨*text_object*⟩s (such as sentences or paragraphs) to pass to the shell for processing.
⟨*text_object*⟩	Any valid *vi* text object definition; can be preceded by a ⟨*count*⟩ parameter (for example, **2}** defines the text object as all text from current position to second following end of paragraph).
⟨*shell_command*(s)⟩	Any valid sequence of one or more HP-UX commands including pipe and tee connections between commands. Standard input for commands is the text object defined by ⟨text_object⟩. ⟨text_object⟩ is replaced by standard output from ⟨shell_command(s)⟩.

Text Replacement: Adjusting Text Paragraphs

As any experienced user of text processors and editors knows, text consisting of choppy lines of varying length are common. Here is and example:

```
This paragraph
consists of several lines of varying length.
It
would look much better if
it was rearranged into a group of lines of more uniform length.
Do you
agree?
```

Shell operations provide a convenient means for easily adjusting paragraphs or other text blocks while in *vi*. While several programs could be used to do this (as well as your own custom shell script), the HP-UX *adjust* command is probably the most convenient.

In its default form, *adjust* arranges contiguous lines of text (in this case taken from standard input provided by *vi*) into a paragraph of successive lines separated by word boundaries and containing the maximum possible number of characters per line up to the default limit of 72 characters per line. If the *adjust* command does not include any non-default options, no extra spaces are provided and the finished paragraph has an even left margin with a ragged right margin.

Here is what *adjust* does to the previous example paragraph following the command **!}adjust** with the cursor placed anywhere on the first line in the paragraph:

```
This paragraph consists of several lines of varying length.  It would
look much better if it was rearranged into a group of lines of more
uniform length.  Do you agree?
```

To obtain even margins on both right and left boundaries, use the **–j** option. Here is the result of the same paragraph using the **!}adjust –j** command:

```
This paragraph consists of several lines of varying length.  It would
look much better if it was  rearranged  into a group of lines of more
uniform length.  Do you agree?
```

The **–m** option is used to change line length. Here is the result of the command **!}adjust –j -m40** or **!}adjust –jm40**. Note the smooth right margin and observe how the options can be separate or combined.

```
This  paragraph  consists  of several
lines of  varying  length.  It  would
look much better if it was rearranged
into a group of lines of more uniform
length.  Do you agree?
```

Here is the result when the **–j** option is omitted:

```
This paragraph consists of several
lines of varying length.  It would
look much better if it was rearranged
into a group of lines of more uniform
length.  Do you agree?
```

Refer to the *adjust*(1) entry in the *HP-UX Reference* for more information about available options. The topics which follow show several other ways for using the *adjust* command.

Adjusting Multiple Paragraphs

Suppose you have four paragraphs of text that are separated by empty lines or lines consisting of paragraph macros. For the purpose of illustration, let us assume that each paragraph contains four sentences (16 sentences in four paragraphs), and the cursor is located somewhere on the first line of the first paragraph.

Since *vi* passes any defined text object to the shell without knowing what you want to do with it, the following two commands are equivalent:

`!4}adjust` **RETURN** (adjust next 4 paragraphs)

or

`!16)adjust` **RETURN** (adjust next 16 sentences)

Either form adjusts all four paragraphs with a left-justified, ragged right margin, up to 72 characters per line. The first command adjusts the next four paragraphs starting with the cursor line. The second adjusts the next 16 sentences starting with the line of the sentence that contains the cursor. When the operation is complete, the cursor is returned to the left margin of the line it was on when the operation started.

Note

When piping text objects to a command, the current cursor position represents the beginning or end of the object sent to the pipeline unless a move command precedes the text object definition sequence. Therefore, be careful when using text objects this way to ensure that the correct text is being sent.

Speeding It Up: Tradeoffs

Whenever you perform an adjust operation this way, *vi* must spawn a new process, load the *adjust* program, run it on the text object, return the result to *vi* and terminate the process. This involves a lot of overhead. Thus it is much faster to adjust 16 paragraphs at once than to adjust the same number of paragraphs, one at a time. However, as the number of paragraphs adjusted in a single operation increases, so does the risk of encountering a text object within the defined text object (such as a table or heading) that you do not want to adjust.

Using Left/Right Shift with Adjust

The shift-right (>>) and shift-left (<<) commands are useful on those occasions where you may need to move the left margin of a text block or paragraph right or left from the left margin or its current position. One method commonly employed by casual users is to insert or delete tabs or spaces at the beginning of each line. However, this can be cumbersome, especially if *adjust* is used to justify the margins (*adjust* has no provision for altering the left margin indent to a given column position).

To solve the problem, consider the following paragraph that was formatted by using the *vi* command `!}adjust -j` RETURN:

```
Occasionally, you may need to move the left margin of a text block or
paragraph right or left from the left margin or its current position.
One method commonly employed by casual users is to insert or delete tabs
or spaces at the beginning of each line. However, this can be
cumbersome, especially if adjust is used to justify the margins (adjust
has no provision for altering the left margin indent to a given column
position).
```

Suppose we need to indent the left margin five columns while holding the same right margin position. First, repeat the previous *adjust* command, but use a right margin of 67 instead of 72 to gain five columns:

```
!}adjust -j -m67

Occasionally, you may need to move the left margin of a text block
or paragraph right or left from the left margin or its current
position. One method commonly employed by casual users is to
insert or delete tabs or spaces at the beginning of each line.
However, this can be cumbersome, especially if adjust is used to
justify the margins (adjust has no provision for altering the left
margin indent to a given column position).
```

Note the narrower paragraph width. To change the default shiftwidth, use the command:

```
:set shiftwidth=5 RETURN
```

then press **7>>** (shift 7 lines right beginning with the current line). The entire paragraph moves to the right with the following result:

```
    Occasionally, you may need to move the left margin of a text block
    or paragraph right or left from the left margin or its current
    position. One method commonly employed by casual users is to
    insert or delete tabs or spaces at the beginning of each line.
    However, this can be cumbersome, especially if adjust is used to
    justify the margins (adjust has no provision for altering the left
    margin indent to a given column position).
```

Shiftwidth is discussed in Chapter 12 entitled Configuring the Vi/Ex Editor.

Text Replacement: Sorting Lists

A common problem in text processing involves taking a random list of items and sorting them in some pre-determined fashion – sometimes one word or item per line, sometimes multiple-column lines sorted according to a certain column, sometimes simply long lines that need to be sorted – and getting the job done with a minimum of effort or difficulty. Normally, this could be a very tedious task. With *vi* it becomes surprisingly simple.

The techniques shown here for sorting are simple. However, they can be readily expanded to fit more sophisticated needs. The examples that follow can serve as a seed bed for more ideas. Refer to Section 1 of the *HP-UX Reference* for more information about the *sort* command.

Sorting the List

Suppose you have the following list of colors somewhere in the middle of a large text file; arranged as indicated, one color per line with one or more empty lines before and after the list (thus forming one paragraph):

```
red
blue
green
orange
yellow
maroon
brown
cyan
purple
chartreuse
violet
crimson
```

First, let us sort the list into alphabetical order, still one color per line, without disturbing surrounding text. Place the cursor on the first line (red), then execute the *vi* command:

```
!}sort RETURN
```

The **!** command tells *vi* that this command is to be sent to the shell for interpretation. The **}** command is a *vi* directive identifying the text object to be processed. It tells *vi* to send the text from the current cursor line through the end of the current paragraph to the shell as standard input for the command which follows. **sort** is the HP-UX command that is to be executed by the shell. Upon completion, the shell returns its standard output from the sort to *vi* which, in turn, uses the processed text to replace the original text that was previously sent to the shell for processing.

As you type the command, notice that when you press !, nothing visible happens (*vi* is holding the character in a buffer, and is waiting for your next command character). The } character represents the text object (the object can be preceded by a count). As soon as *vi* recognizes a valid text object, the ! character is displayed at the bottom of the screen on the command line, but the object is never displayed just as in normal *vi* editing operations. Once the text object has been defined, the remaining characters in the command are treated as a valid shell command, and are sent directly to the shell for interpretation. After processing by *vi*, only the following characters from the command sequence are visible on the command line at the bottom of the terminal screen:

```
!sort
```

The result that replaces the original text is:

```
blue
brown
chartreuse
crimson
cyan
green
maroon
orange
purple
red
violet
yellow
```

Working with Multi-Column Lines

Occasionally, you may have a similar series of lines except that each line contains several columns of text separated by tabs or other whitespace/delimiter characters. You may want to sort the lines based on the contents of the first column of each line. However, you may also need to sort the lines based on the contents of column 3 or 4, for example, or based on contents from any position in the line to any other position in the same line. For more information, refer to the *sort*(1) manual page entry in the *HP-UX Reference*, and to discussions of the *sort* command elsewhere in this volume.

For example, to sort a paragraph containing several lines, each of which contains five columns, in alphabetical order according to the contents of column 2, and ignoring uppercase/lowercase differences, use:

```
!}sort -f +1 -2
```

sort is explored in greater detail among the text processing topics discussed elsewhere in this volume.

Text Replacement: Rearranging Lists into Tables

Another common text processing problem entails rearranging a single column of text or data, one item per line, into a multiple-column table format. The HP-UX *pr* command provides an easy way to do this. For example, using the unsorted list of colors from the previous section, execute the following *vi* command:

```
!}pr -4t RETURN
```

while the cursor is on the first color (red). This single-column list:

```
red
blue
green
orange
yellow
maroon
brown
cyan
purple
chartreuse
violet
crimson
```

is replaced by this four-column layout:

```
red             orange          brown           chartreuse
blue            yellow          cyan            violet
green           maroon          purple          crimson
```

However, the list is not sorted into alphabetical order. To sort and format the list in a single operation, execute both commands in a single pipeline as follows:

```
!}sort | pr -4t
```

The resulting table is now in alphabetical order:

```
blue            crimson         maroon          red
brown           cyan            orange          violet
chartreuse      green           purple          yellow
```

As discussed previously, the *sort* part of the command sequence sorts the paragraph into alphabetical order in a single-column format. The *pr* command option **−4** specifies four-column output (columns are separated by an appropriate combination of tab and space characters); the **t** option suppresses page headers, footers, and any empty lines that would normally be created before or after the processed text when formatting printed pages. The processed output text from *pr* is returned to *vi* by the shell so it can be used to replace the original text object.

However, the embedded tab characters in the formatted multi-column output may not be acceptable for some situations. Here is what the text really looks like when the tabs are identified:

```
blue<tab ><tab >  crimson< tab  >    maroon<tab >     red
brown<tab><tab >  cyan<tab ><tab>    orange<tab >     violet
chartreuse<tab >  green<tab><tab>    purple<tab >     yellow
```

where <tab > (including spaces within the angle brackets < and > represents the space on the terminal screen that is consumed by the corresponding tab character in the sorted output.

Expanding Tabs to Spaces in Columnar Output

If the table must not contain tab characters, they are easily eliminated by using another HP-UX command, *expand*, in the pipeline:

```
!}sort | pr -4t | expand RETURN
```

As before, the *sort* part of the command sequence sorts the specified text object into alphabetical order in a single-column format; *pr* formats the sorted output into four-column output with columns separated by tab characters, and *expand* converts each tab character into a string of blanks to create the correct appearance on the terminal screen (*pr* and *expand* use the same default column numbers for tab stop positions). The shell returns the processed result to *vi*:

```
blue          crimson       maroon        red
brown         cyan          orange        violet
chartreuse    green         purple        yellow
```

without any embedded tab characters.

Adding tbl Macros

The table preprocessor *tbl* converts table source text in a special form into a series of commands sequences that can be used in conjunction with an *nroff* or *troff* text formatting program or *mm* (memorandum macro) processing program. The output from the *sort | pr* command shown previously would not be suitable for use with *tbl* (table formatter) preprocessor macros because *tbl* expects a single tab character as a field separator between items on each line.

In both examples shown earlier, the table contains either multiple adjacent tab characters and spaces, or multiple spaces between items. Therefore, both are incompatible with *tbl* unless some changes are made. To convert a combination of multiple tabs and spaces to the single tab separator expected by *tbl*, use a global substitution technique similar to the following (cursor is anywhere on the first of the three lines in the table) on the unexpanded table:

```
:.,.+2s/[tab][tab]*[ ]*/tab/g
```

where *tab* is a single tab character.

To convert each occurrence of a series of one or more spaces (as in the second example where tabs were expanded to spaces) to a single tab, use a construction like this:

```
:.,.+2s/[ ][ ]*/[tab]/g
```

Either method applied to the appropriate text produces the following (source text contains no leading blanks or tabs):

```
blue<tab>crimson<tab>maroon<tab>red
brown<tab>cyan<tab>orange<tab>violet
chartreuse<tab>green<tab>purple<tab>yellow
```

where `<tab>` represents a single tab character. This format is now compatible with *tbl* formatting requirements. To arrange the three lines into a boxed table with each pair of adjacent columns separated by a single vertical line add three lines before and one line after as follows:

```
.TS
center box;
l | l | l | l .
blue<tab>crimson<tab>maroon<tab>red
brown<tab>cyan<tab>orange<tab>violet
chartreuse<tab>green<tab>purple<tab>yellow
.TE
```

As always, the table definition sequence on the second and third line could be changed to add more information such as column headings, etc.

Sorting by Field before Formatting in Columns

You may occasionally need to sort a list consisting of two or more words (such as first and last names in a name list) based on the second (or later) column. For example, the following list of notable names:

```
George Washington
Henry Clay
John Adams
Napoleon Bonaparte
Abraham Lincoln
John Calvin
Martin Luther
```

is easily sorted into three columns by placing the cursor anywhere on the first line in the list then using the command:

```
!}sort -t\  +1b |pr -3t
```

```
John Adams<tab><tab>Henry Clay<tab><tab>Martin Luther
Napoleon Bonaparte<tab>Abraham Lincoln<tab><tab>George Washington
John Calvin
```

The **−t**⟨*space*⟩ option defines the ASCII space character between first and last names as the field separator. A backslash precedes the space to protect it from interpretation by the shell. A second space separates the **−t** option from the **+1b** option which tells *sort* to arrange entries according to the contents of the first field following the left or first field on the line, thus sorting by surname. (A bug in *pr* may place an unwanted tab character between first and last name on some names in the list.)

Text Insertion: Reading Shell Output

Suppose you are writing a procedure for how to use a new program you have written and need to include an example of expected program output. Obviously, you want the result to be accurate. What better way then than to let the program provide the result and place it directly into the file you are editing instead of sending it to the standard output which usually shows up on the terminal display screen?

In an earlier chapter we talked about merging a file into text using the **:r** command which reads a file into the buffer file starting on a new line following the current cursor line. Suppose we replace the filename (following the space after the **r** command) with the shell escape character (**!**) and an HP-UX command sequence as follows:

:r !⟨*hpux_command*⟩ RETURN

Voila! Standard output appears in the file right where the file normally goes when a filename is specified. Now for an example:

Suppose you have three files, each containing ten items, one item per line. *file1* contains the letters **a** through **j**, *file2* contains the numbers **0** through **9**, and *file3* contains the words **zero** through **ten** (note that *file3* contains 11 lines instead of ten like the other two files). With the cursor in the current line of visible text followed by another visible text line, the command:

```
:r !paste file1 file2 file3 RETURN
```

produces the following result:

```
the current line of visible text
a    0       zero
b    1       one
c    2       two
d    3       three
e    4       four
f    5       five
g    6       six
h    7       seven
i    8       eight
j    9       nine
             ten
another visible text line
```

Note

There is a big difference between:

:w! *filename*

and

,image **:w** **!***hpux_command*

To reinforce the difference, the command form:

:r **!***hpux_command*

is emphasized instead of the equivalent **;r!** *hpux_command* which produces identical results for read operations only.

Check Your Spelling the Easy Way

One perplexing problem for typists and terminal users and humans in general is having an easy way to check for spelling and typing errors. This becomes very easy when using *vi* in an HP-UX environment, and as usual, there are several ways of doing it. The method shown here is simple and effective, and is not significantly more cumbersome than using specialized programs to accomplish the same thing.

To check spelling on the entire file, type the command:

```
:w !spell >spell.errors RETURN
```

When the prompt to continue is displayed upon completion, press any key to restore the display screen and continue. Misspelled words are now stored in the file named *spell.errors*. However, it is a bit of a nuisance to have them in an external file, so let's import the errors file to the working file being edited. Type **G** to move the cursor to the last line in the file, then type:

```
:r spell.errors RETURN
```

to read the errors file in at the end of the file. If you prefer to have the errors at the beginning of the file, place the cursor on the first line of the error file text, type the command **dG**, then type **1G** to move the cursor to the beginning of the file and type **P** to insert the deleted text before the first line.

Now it is a simple matter to scan through the merged errors file, delete those lines that may be correctly spelled but were not included in the HP-UX spelling dictionary database. After eliminating the correctly spelled words that showed up as misspelled, you can search for the others, make the needed corrections, then delete each entry from the errors list after it has been corrected and continue with the next error.

Writing to a Shell Command Instead of a File

The previous spelling example shows a typical situation where a block of text needs to be written as standard input to a shell command without bringing the standard output back to replace the original text object. The command used is deceptively similar to the standard forced **write** command (**:w!**) except for the placement or absence of a blank (space or tab) before the exclamation point. Command format to force the buffer to be written to *filename*, even if the file already exists (provided you have write permission) is:

`:w!` *filename* RETURN

To send the entire file as standard input to an *hpux_command*, use:

`:w !`*hpux_command* RETURN

The type of output produced by the command (or shell script or other program) and its destination is determined by the command and what options, if any, are used with it.

As discussed in the section on using the **write** command in Chapter 9, text blocks defined by file markers, text object specifications, etc. can be sent to the command instead of the entire file by preceding the **w** with marker names, line numbers, *ex* addresses, or any other compatible construction that can be used with the **w** command.

Custom Processing

The **:w !** construction can also be used to process the file through a shell script, *awk* script, or other device or program as your needs might dictate. No further discussion of such devices is presented here because the level of competence required for writing such scripts is such that the information presented earlier in this topic is sufficient for providing the standard input text.

Editor: Configuring the Vi/Ex Editor 12

For most casual users, the default configuration of *vi* is quite adequate. However, you may have some special needs or preferences that make it advantageous to have certain editor operating characteristics and features changed to suit your situation. You have two choices:

- Use the **:set** command to immediately change the desired operating characteristic or feature, or
- Store the desired characteristics and features in a default configuration file that *vi*/*ex* uses each time a new session begins.

The first part of this chapter explains how to use configuration commands to set up characteristics and features. The latter part of the chapter explains how to set up a configuration file so that your preferred characteristics are automatically configured every time you use *vi*.

Configuration Options

The following operating options can be configured by use of the *vi/ex* **:set** command. Option listing shows typical default values for most releases of *vi*. To determine the current options settings for your session, use the **:set all** command. Commands are in (almost) alphabetical order if you ignore the no prefix on those options that are disabled.

Typical Default *vi/ex* Options Settings

```
noautoindent            nonovice                noshowmode
autoprint               nonumber                noslowopen
noautowrite             nooptimize              tabstop=8
nobeautify              paragraphs=IPLPPPQPP LIbp taglength=0
directory=/tmp          prompt                  tags=tags /usr/lib/tags
noedcompatible          noreadonly              term=hp
noerrorbells            redraw                  noterse
flash                   remap                   timeout
hardtabs=8              report=5                ttytype=hp
noignorecase            scroll=11               warn
nolisp                  sections=NHSHH HU       window=8
nolist                  shell=/bin/sh           wrapscan
magic                   shiftwidth=8h           wrapmargin=0
mesg                    noshowmatch             nowriteany
nomodelines
```

Enabling, Disabling, and Setting Options

To enable or disable an option, use the **:set** command followed by the option name then press RETURN as follows:

:set *option_name* RETURN	Enable the option,
:set no*option_name* RETURN	Disable the option, or
:set *option_name=value* RETURN	Assign a value to the option.

Some (but not all) commands can be abbreviated to save typing. The abbreviation for each option, if available, is listed with the command in the following list which describes each option in greater detail. When typing a command, spell the option name in its entirety as shown above, or use the abbreviation as shown in the paragraphs that follow. No other spellings are recognized, and produce an error if used.

Option Descriptions

The following topics describe the options that are supported on *vi* and *ex*. Each option is recognized by one editor or the other or both as indicated between parentheses in the heading for that option. In general, *vi* options apply equally to *view*, and *ex* options apply to *edit* and *vedit*.

autoindent (vi/ex)

abbr: **ai** Default: **noai**

To enable:	**:set autoindent**	or	**:set ai**
To disable:	**:set noautoindent**	or	**:set noai**

Automatic indenting is most commonly used when writing structured programs as in C or Pascal. When this option is set and the editor is in insert mode, the editor determines the current indent (as it exists in the preceding line when a new line is started), then uses that indent on all subsequent lines until it is changed. Thus, when starting a new line, if the previous line starts with its first visible character in column 10 from the left side of the display screen, all subsequent lines will start in the same column 10.

If disabled, RETURN moves the cursor to the extreme left margin of the next line, regardless of the indent of current or previous lines.

How Vi/Ex Determines Current Indent

Current indent is defined as the column position of the first visible character on the line where the cursor was located at the time insert mode is entered. If an **O**pen command is used to open a new line before the current line, the current line indent is used; not the previous line. Note that *vi/ex* uses a combination of tabs and spaces to set the indent of new lines being added. If your requirements are such that tabs are not acceptable, this option probably should not be used.

Changing Current Indent

To change autoindent on a given new line, space over to the desired column to increase indent. To decrease indent to the previous **shiftwidth** column, use CTRL-D (back-tab) as the first character in the line. To input a single line with no indent then return to the previous indent, use a circumflex (^) followed by CTRL-D at the beginning of the unindented line. If the first character pressed after RETURN is not a CTRL-D or ^ followed by CTRL-D, no change in indent occurs. However, if you start a new line with one or more tabs or spaces, the next following line is started at the new indent determined by the position of the first visible character on the current line.

No Whitespace Added to Empty Lines

Autoindent automatically adds indenting whitespace (space or tab) characters whenever text is added in the line. If an empty line (such as between paragraphs or blocks of program source code) is created, no whitespace characters are added to that line in the file (an empty line has zero length except for its terminating newline character).

When **autoindent** is disabled, RETURN moves the cursor to the left column of the next line, regardless of the indent in previous lines.

Autoindent is inactive during *global* commands and when the keyboard input device is not a terminal.

autoprint (ex only)

abbr: **ap** Default: **ap**

To enable:	**:set autoprint**	or	**:set ap**
To disable:	**:set noautoprint**	or	**:set noap**

Automatically prints the current line after each *delete*, *copy*, *join*, *move*, *substitute*, *l*, *undo*, or *shift* command. Has the same effect as supplying a trailing **p** at the end of each such command. *Autoprint* is inactive during global operations, and only applies to the last command on the line when multiple commands are combined on a single line. This option is not used by *vi*.

autowrite (vi/ex)

abbr: **aw** Default: **noaw**

To enable:	**:set autowrite**	or	**:set aw**
To disable:	**:set noautowrite**	or	**:set noaw**

Causes the contents of the buffer to be written to the current file if it has been modified and a *next*, *rewind*, *tag*, or ! (shell-escape) command is encountered in *vi*[1] or *ex*, or a ^ (switch files) or ^| (tag goto) command in *vi*. Note that the *edit* and *ex* commands do not force an autowrite.

[1] Commands must be preceded by a colon when using *vi*.

Bypassing the Autowrite Feature

You may occasionally encounter times when you have been editing a file then decide to abandon the file without placing the modified text back in permanent storage. If the autowrite option is set (which means that if you try to quit the editor it will write the file back anyway; not what you want) you can prevent autowriting when terminating or changing files by using alternate forms of standard editor commands:[1]

- **quit!** instead of **quit**,
- **edit** instead of **next**,
- **rewind!** instead of **rewind**,
- **stop!** followed by the **tag!** command instead of **tag**,
- **shell** instead of **!** (when you exit from the new shell, the edit resumes).

From *vi*, use:

- **:e#** when switching between two files or
- **:ta!** command when using tag files to find text segments.

beautify (vi/ex)

abbr: **bf** Default: **nobeautify**

To enable:	**:set beautify**	or	**:set bf**
To disable:	**:set nobeautify**	or	**:set nobf**

Setting this option discards all control characters except tab, newline, and form-feed during input. A complaint is registered the first time a backspace is discarded. *Beautify* affects only keyboard text input. It does not alter command input.

directory (vi/ex)

abbr: **dir** Default: **directory=/tmp**

Specifies which directory is used by *vi* or *ex* when creating the buffer file following an **edit** *file* command from within the editor. This option does not affect the buffer location if the option is set during the session. To control the location of the buffer at the opening of each session, the option must be set in the *.exrc* file in your home directory (*$HOME*) or the session must be opened without specifying a filename. If you open the session without specifying a filename on the HP-UX command line, set the option after the session is open, then specify the name of the file to edit, the buffer location will be determined by the directory specifed when this option was set.

If write permission is not available for the specified directory and the directory is specified by *.exrc*,, the editor exits and terminates immediately. If the file is being specified interactively after the session is open and the directory is specified and write permission is not available in that directory, an error message is generated.

edcompatible (vi/ex)

abbr: **ed** Default: **noedcompatible**

To enable:	**:set edcompatible**	or	**:set ed**
To disable:	**:set noedcompatible**	or	**:set noed**

When this option is enabled:

- If a **g** (global) or **c** (check) suffix is present on a *substitute* command, the **global** or **check** flag is toggled and the command is processed accordingly. If the suffix is absent on subsequent substitution commands, the toggled flag (unchanged from the previous substitution command) is used to determine how to process the command. When a new suffix appears on a *substitute* command, the flag is again toggled to its opposite state and processing is reversed accordingly.
- An **r** suffix on a substitution command recognizes the *ed* metacharacter **%** as the replacement string from the last preceding substitution instead of using the ~ metacharacter that is normally used in *vi/ex* for that purpose. As usual, the **&** metacharacter represents the text string that matched the search regular expression in the current substitution string.

errorbells (vi/ex)

abbr: **eb** Default: **noerrorbells**

To enable:	**:set errorbells**	or	**:set eb**
To disable:	**:set noerrorbells**	or	**:set noeb**

When enabled, this option precedes error messages with a bell **only** on terminals that do not support a standout or highlighting mode such as inverse video. If the terminal supports highlighting, the bell is never used prior to error messages and this option has no effect.

flash (vi/ex)

abbr: none Default: **flash**

To enable:	**:set flash**	or	**:set fl**
To disable:	**:set noflash**	or	**:set nofl**

When enabled, this option causes the screen to flash instead of beeping, provided an appropriate *flash_screen* entry is present in the *terminfo* data base for the terminal being used (see *terminfo*(4) entry in *HP-UX Reference* for more information).

hardtabs (vi/ex)

abbr: **ht** Default: **hardtabs=8**

To enable:	**:set hardtabs**	or	**:set ht**
To disable:	**:set nohardtabs**	or	**:set noht**

Defines the spacing between hardware tab settings and the number of spaces used by the system when expanding tab characters. Tab stops are placed in each column number (starting at the left edge of the screen) that corresponds to an integer multiple of the numeric value used when setting this option.

ignorecase (vi/ex)

abbr: **ic** Default: **noignorecase**

To enable:	**:set ignorecase**	or	**:set ic**
To disable:	**:set noignorecase**	or	**:set noic**

When enabled, this option maps all uppercase characters in text to lowercase when matching regular expressions. It also maps all uppercase characters in regular expressions to lowercase except for character-class specification characters.

lisp (vi/ex)

(no abbr) Default: **nolisp**

To enable:	**:set lisp**
To disable:	**:set nolisp**

When this option is set, it causes the *Autoindent* option to indent appropriately for *lisp* program code and modifies the meaning of ( ), { }, [[, and]] commands in *open* and *vi* so that they correspond to *lisp* usage.

list (vi/ex)

(no abbr) Default: **nolist**

To enable:	**:set list**
To disable:	**:set nolist**

Setting this option causes all printed lines to be displayed less ambiguously, showing tabs and newlines as in the *ex* **:list** command.

magic (vi/ex)

(no abbr) Default: **magic** for *ex* and *vi*

To enable: **:set magic**
To disable: **:set nomagic**

If nomagic is set, the number of regular expression metacharacters is greatly reduced, with only ^ and **$** having special effects. In addition, if the metacharacters ~ (text used in the last previous replacement) and **&** (text matching the regular expression used in the current replacement) appear in the replacement pattern, they are treated as normal characters and must be preceded by a \ if they are to be used as special characters. Any metacharacters that have been disabled by setting nomagic can re-enabled for use as metacharacters while nomagic remains set by preceding them with a \. The principle metacharacters affected by this option are: ., [(and the accompanying]), **&**, ~ and *.

mesg (vi/ex)

(no abbr) Default: **mesg**

To enable: **:set mesg**
To disable: **:set nomesg**

Setting **nomesg** blocks write permission to your terminal from other system users while you are using *vi* (equivalent to the HP-UX command **mesg n** RETURN). If *mesg* is set, other users can use the HP-UX *write* command to send messages to your terminal, possibly disrupting the screen display unless you have disabled that ability from a **mesg n** command prior to starting the editor.

modelines (vi/ex)

abbr: **modeline** Default: **nomodelines**

To enable:	**:set modelines**	or	**:set modeline**
To disable:	**:set modelines**	or	**:set nomodeline**

If **modelines** is set, the editor scans the first and last five lines in the file when a new file is opened, looking for any *ex* commands that might exist in those lines. After all *.exrc* and EXINIT commands are processed, and before editing control is given to the user, the commands embedded in the first and last five lines of the file, if they exist, are executed.

Any commands that are placed in the file must lie within the first and/or last five lines of the file, must be prefixed by **ex:** or **vi:** and must be terminated by a colon (**:**); all in a single line. The commands can be prefixed by **ex:** or **vi:**, but only valid *ex* editor commands can be used. For example, to set the list option, use the command:

ex: set list:

or

ex: set list:

anywhere in the first or last five lines. There is no restriction on the length of the command except that it must all fit on a single line and not exceed approximately 1020 total characters in the line. To separate multiple commands on a single modeline, use the vertical bar character (|).

Be careful when using this option to make sure that the first and last lines in a normal file cannot be incorrectly interpreted as commands. To be safe, your *.exrc* file should probably leave this option disabled unless you have a specific need that requires enabling modelines.

number (vi/ex)

abbr: **nu** Default: **nonumber**

To enable:	**:set number**	or	**:set nu**
To disable:	**:set number**	or	**:set nonu**

Setting this option causes the editor to precede all printed or displayed lines of text with a line number. In text input mode (insert, append, or open) a line number is provided and the cursor advanced to the beginning text column whenever a new line is started. Line numbers are displayed on the terminal only; they are not included in the file when it is written back to permanent storage. To add line numbers to a file, use the HP-UX *pr* command.

optimize

abbr: **opt** Default: **optimize**

To enable:	**:set optimize**	or	**:set opt**
To disable:	**:set nooptimize**	or	**:set noopt**

Setting this option suppresses automatic carriage returns by the terminal on terminals that do not support direct cursor addressing. This streamlines text output in certain situations such as when printing multiple lines that contain leading white space.

paragraphs (vi/ex)

abbr: **para** Default: **paragraphs=IPLPPPQPP LIbp**

Specifies the one- or two-character macro names that are to be recognized (in addition to empty lines) as paragraph boundaries when interpreting cursor movements related to the { and } commands in *vi* or in *ex* **open** mode. All macros defined by the **sections** option are also recognized as paragraph boundaries in addition to those defined by **paragraphs**.

[1] A macro is a one- or two-character symbol following a period (.) at the beginning of a line of text that serves as a formatting command to a text formatting program such as *nroff*. The macro usually replaces a larger set of lower-level commands that would be necessary to, for example, clean up a paragraph, space down for the next paragraph, then continue with the next block of text, or perform some other comparable task.

If any macros have a single-character name, use a blank (space character) to substitute for the missing second character in the name. When typing a space character in such situations, the space must be preceded by a backslash (\) to prevent the editor from interpreting it as a delimiter.

Example: To define recognized paragraph macros to include **.bullet**, **.item**, **.step**, and **.note** in addition to blank lines and the *mm* macros **.P** and **.PP**, use the command:

```
set paragraphs=PPP\ buitstno RETURN
```

after a colon (or colon prompt from *vi*) or in the *.exrc* file in your home directory *$HOME*. When the **:set all** command is used to list the current options, the backslash preceding the space is not shown because it is consumed during initial interpretation by the editor.

When the editor is scanning for paragraph boundaries, only the first two letters after the period at the beginning of the line are used by *vi* to recognize a macro. Any subsequent characters on a text line containing a defined macro character pair are ignored by *vi* and *ex*.

prompt (ex only)

(no abbr) Default: **prompt**

To enable: **:set prompt**
To disable: **:set noprompt**

If this option is set, the editor prompts for a new command when in command mode by printing a colon.

readonly (vi/ex)

abbr: **ro** Default: **noreadonly**

To enable:	**:set readonly**	or	**:set ro**
To disable:	**:set noreadonly**	or	**:set noro**

This option sets the read-only flag for the file being edited, thus preventing accidental overwriting at the end of the session. This option is equivalent to invoking *vi* or *ex* with the **−R** option or using the HP-UX *view* command. Setting this option in the *.exrc* file in your home directory or in the *EXINIT* variable in your *.profile* file has the effect of making all files being edited read-only so that the edited result must be placed elsewhere. It does not, however, prevent overwriting the original file by using the **:w!** command.

redraw

(no abbr) Default: **noredraw**

To enable:	**:set redraw**
To disable:	**:set noredraw**

When this option is set, the editor simulates an intelligent terminal on a dumb terminal. This is usually accomplished by outputting new characters on the current line to the right of the cursor position or reprinting subsequent lines on the screen as needed when inserting, deleting, or changing the number of visible characters on the display. This method usually results in large amounts of data being transferred to the terminal, and is useful only when the terminal-to-computer data path operates at very high speed.

remap (vi/ex)

(no abbr) Default: **remap**

To enable:	**:set remap**
To disable:	**:set noremap**

If this option is set, macro redefinitions are repeatedly followed until the last definition is found. For example, suppose **o** is mapped to **O**, and **O** is mapped to **I**. If *remap* is set, **o** maps to **I**; if *noremap* is set, the link between **O** and **I** is ignored and **o** instead maps to **O** as originally defined.

report (vi/ex)

(no abbr) Default: **report=5**

To enable: **:set report**
To disable: **:set noreport**

Specifies a threshold for feedback from commands. Any command that 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 (vi/ex)

(no abbr) Default: **scroll=11**

To enable: **:set scroll**
To disable: **:set noscroll**

Determines the number of logical lines scrolled when an end-of-file character (typically CTRL-D unless redefined otherwise) is received from the terminal keyboard while in command mode. The value also defines the number of lines printed by a command-mode command (double the value of *scroll*).

sections (vi/ex)

(no abbr) Default: **sections=SHNHH HU**

Specifies the one- or two-character macro[1] names that are to be recognized as section boundaries when interpreting cursor movements related to the [[and]] commands in *vi* or in *ex* **open** mode. If any macros have a single-character name, use a blank (space character) to substitute for the missing second character in the name.

If any macros have a single-character name, use a blank (space character) to substitute for the missing second character in the name. When typing a space character in the macro names that are to be recognized as section boundaries, the space must be preceded by a backslash (\) to protect it from interpretation as a delimiter by the editor.

Example: Suppose you are using a proprietary text formatting program where section heads are set by the section level macros **.1** through **.4**. To reconfigure the editor so it can recognize the macros **.1**, **.2**, **.3**, and **.4**, use the command:

```
set sections=1\ 2\ 3\ 4\ RETURN
```

after a colon (or the *ex* colon prompt) or in the *.exrc* file in your home directory *$HOME*.

shell (vi/ex)

abbr: **sh** Default: **sh=/bin/sh**

This option specifies the path and filename of the user shell that is to be used when a shell escape command (!) or *sh* command is encountered and the user SHELL environment variable is undefined. If the SHELL environment variable is defined, that value is used instead.

[1] A macro is a one- or two-character symbol following a period (.) at the beginning of a line of text that serves as a formatting command to a text formatting program such as *nroff*. The macro usually replaces a larger set of lower-level commands that would be necessary to, for example, clean up a paragraph, set a new section head, then continue with the next paragraph, or perform some other comparable task.

shiftwidth (vi/ex)

abbr: **sw** Default: **shiftwidth=8**

To enable:	**:set shiftwidth**	or	**:set sw**
To disable:	**:set noshiftwidth**	or	**:set nosw**

Specifies the spacing between software tab stops. This value is used when: (a) reverse tabbing with **^D**, (b) using **autoindent** to append text, and (c) using the right/left shift (>> and <<) commands.

showmatch (vi/ex)

abbr: **sm** Default: **noshowmatch**

To enable:	**:set showmatch**	or	**:set sm**
To disable:	**:set noshowmatch**	or	**:set nosm**

If this option is set and you are in *vi* or using *ex* in *open* mode, when a) or } is typed, the cursor moves to the matching (or { for one second (if the matching character is on the screen) then returns. This feature is very useful when working with the Lisp programming language but a general nuisance otherwise.

showmode (vi only)

(no abbr) Default: **noshowmode**

To enable:	**:set showmode**
To disable:	**:set noshowmode**

If this option is set and you are in *vi*, the message `INPUT MODE` is displayed in the lower right-hand corner of the *vi* display area of the terminal screen whenever *vi* is operating in input mode. This feature is very helpful for beginning users who may experience difficulty in understanding the difference between command mode and input mode and knowing which mode is currently active.

The `INPUT MODE` message displayed by *vi* relates only to *vi* program operation. When *vi* is being used on intelligent terminals, *vi* may overwrite the current screen or place the terminal in terminal screen input mode, whichever requires less communication overhead. When the terminal is placed in its own internal input mode, it may display a second `Input Mode` message, usually below the softkey labels. The terminal input-mode message has no direct relationship to the `INPUT MODE` message displayed by *vi* when the showmode option is enabled.

slowopen (vi/ex)

abbr: **slow** Default is terminal- and speed-dependent

To enable:	**:set slowopen**	or	**:set slow**
To disable:	**:set noslowopen**	or	**:set noslow**

Setting this option alters the display algorithm used for *vi* editing to accommodate slow or unintelligent terminals by limiting the printing of input or new text in exchange for better operating speeds.

tabstop (vi/ex)

abbr: **ts** Default: **tabstop=8**

This option defines the tab spacing used by the editor when expanding tabs in the output file to match display tabstop boundaries.

taglength (vi/ex)

abbr: **tl** Default: **taglength=0**

Specifies the maximum number of characters in a tag that are to be treated as significant. Characters beyond the limit are ignored. A value of zero (default value) means that all characters in the tag are significant.

tags (vi/ex)

(no abbr) Default: **tags=tags /usr/lib/tags**

Specifies path and file names to be used as tag files for the *tag* command or **−t** option when the editor is started. *vi* (or *ex*) sequentially searches the specified tag files for the tag name, then uses the tagfile entry to open the file containing the tagged text and search for the tag in that file. Default tag file searching begins in file *tags* in the current directory (if present), then proceeds to the master system-wide tags file in */usr/lib*. Tags are most commonly used when editing large complex program structures that involve a large number of files in multiple directories, although they are useful in much smaller structures.

term (vi/ex)

(no abbr) — Value obtained from the environment variable, TERM

Defines the type of terminal being used with the editor. Value is obtained from the TERM user environment variable and cannot be altered from within *vi/ex*.

terse (vi/ex)

(no abbr) — Default: **noterse**

To enable: **:set terse**
To disable: **:set noterse**

Setting this option specifies shorter error diagnostics for the experienced user.

timeout (vi/ex)

(no abbr) — Default: **timeout**

To enable: **:set timeout**
To disable: **:set notimeout**

This option sets or disables the timer used to determine whether an escape character is the escape key (such as when [ESC] is used to terminate input mode in *vi*) or the first character in a two-character escape sequence representing arrow keys, function keys, etc. If set, the timeout function is enabled, meaning that if an escape character is not followed within the time limit by another character, the escape is treated as a separate character rather than as part of a two-character sequence.

If this option is disabled (**:set notimeout**), the timeout counter is disabled and any escape character received is always treated as the first character in a two-character escape sequence. Length of the timeout period and its effect on typing speed is discussed in Chapter 2.

ttytype (vi/ex)

(no abbr) Value obtained from the environment variable, TERM

Defines the ttytype for the terminal being used with the editor. Value is obtained from the TERM user environment variable and cannot be altered from within *vi/ex*.

warn (vi/ex)

(no abbr) Default: **warn**

To enable: **:set warn**
To disable: **:set nowarn**

Warn if there has been "no write since last change" before a **!** or *shell* command escape. **nowarn** disables the message.

window (vi/ex)

(no abbr) Default: **window=speed dependent**

Specifies the number of lines that are displayed in a *vi* text window when a file is opened or after a jump to another location in the file, based on modem baud rate. Default is 8 lines at slow speeds (600 baud or less), 16 at medium speed (1200 baud), and the full screen minus one line at higher speeds. The limited number of lines only applies when the entire screen must be redrawn. If you are working in a limited area in the file, the number of lines displayed increases until the screen is full. The screen then remains full until an editor command creates a situation where the entire screen must be redrawn.

This option is useful when slow-speed telephone line modems are used for remote terminals to improve screen updating performance by restricting the amount of display area that must be altered during redraws, scrolls, etc.

w300, w1200, w9600 (vi only)

(no abbr) Default: **not invoked**

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 use in an EXINIT variable or *.exrc* file, and make it easy to change the 8-line/16-line/full-screen rule.

wrapscan (vi/ex)

abbr: **ws** Default: **wrapscan**

To enable:	**:set wrapscan**	or	**:set ws**
To disable:	**:set nowrapscan**	or	**:set nows**

When this option is set, pattern searches resulting from a **/**, **?**, **n**, or **N** command automatically wrap around to the opposite end of the file and continue whenever beginning- or end-of-file is reached.

wrapmargin (vi/ex)

abbr: **wm** Default: **wrapmargin=0**

To enable:	**:set wrapmargin**	or	**:set wm**
To disable:	**:set nowrapmargin**	or	**:set nowm**

Defines the position of the right margin with respect to the right-hand screen boundary that is to be used for automatically wrapping to a new line when the margin is exceeded during text input (automatic newline insertion) from *vi* or from *open* mode. See the Continuous Text Input topic in Chapter 2 for more details.

writeany (vi/ex)

abbr: **wa** Default: **nowriteany**

To enable:	**:set writeany**	or	**:set wa**
To disable:	**:set nowriteany**	or	**:set nowa**

Enabling this option inhibits the checks normally made before *write* commands, thus allowing a write to any file that the system protection mechanism will allow.

Automating Editor Configuration

There are three ways to automatically configure the editor each time you use it with the options, macro definitions, and other characteristics that you prefer:

- Define non-default values in your local environment variable EXINIT, or
- Build an editor configuration file named *.exrc* in your home directory and/or in the current directory that contains a series of *ex* editor **:set** commands, macro definitions and other desired characteristics.
- Embed *ex* editor commands in the first and/or last five lines of the file being edited.

Each time the HP-UX command that starts the *vi/ex* editor is executed, the editor searches for the environment variable EXINIT and uses its contents as configuration commands if it exists. If EXINIT is not defined, the editor then searches for the file *.exrc* in your home directory and uses its configuration commands if the file exists. If neither EXINIT nor *.exrc* exist, the default values discussed earlier in this chapter are used instead.

After processing the EXINIT variable and/or .exrc file in your home directory, the editor then searches the current directory for another file named *.exrc*. If the file exists, it is also processed. This provides a means to have various kinds of files in assorted directories and alter the editor configuration to meet the needs of each directory.

After completing the above tasks, the editor opens the file being edited then, if the **modelines** option is set, it scans the first and last five lines in the file to determine whether any *ex* commands have been placed there. If so, the commands are executed before editing control is transferred to the user. See the **modelines** option discussion earlier in this chapter for more information about the modelines option.

An example *.exrc* file that can be used as a starting base is contained in file */etc/d.exrc*. Here is a modified form of that file with some features you might find interesting. Lines beginning with a double quote character are comments and are ignored by the editor during start-up. Their presence slows the start-up process somewhat, but they make the script easier to interpret.

```
"
" This .exrc file contains comments to tell you what each command
" does. Some commands may not be desirable in your case.  If so,
" comment them out by using the " character, or delete them.
"
" The actual names of special keys are in capital letters (like INSERT LINE).
" The exact names are not universally used, but you should have no trouble
```

```
" figuring out which keys they refer to.  Some may not be present on
" your terminal, depending on make and model.
"
" Caution: this file reflects characteristics of an hp2622 terminal
"          or terminal emulator ($TERM=hp2622). If you use other
"          terminals or other $TERM values, changes may be needed.
"
"   $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
"   $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
"
"    Set up automatic indenting and set right margin at column 72:
"
set autoindent wrapmargin=8
"
"    Block any messages from other users to protect display:
"
set nomesg
"
"    Change HOME UP key to move to upper left corner of screen:
"
map  ^[h  H
"
"    Change HOME DOWN key to move to lower left corner of screen:
"
map  ^[F  L
"
"    Change Left Arrow key to BACKSPACE:
"
map  ^[D ^H
"
"    Change NEXT PAGE key to CTRL-F (ACK) command character:
"      (Cannot be used in certain situations)
"
map  ^[U ^F
"
"    Change PREV PAGE key to CTRL-B command
"
map  ^[V ^B
"
"    Change ROLL DOWN key to Scroll window down 1, move cursor up 1 line:
"
map  ^[T ^Yk
"
"    Change ROLL UP key to Scroll window up 1, move cursor down 1 line:
"
map  ^[S ^Ej
"
"    Change INSERT CHAR key to vi insert (i) command:
"
map  ^[Q  i
```

```
"
"    Change DELETE CHAR key to vi delete character (x) command:
"
map  ^[P  x
"
"    Change DELETE LINE key to vi delete line (dd) command:
"
map  ^[M  dd
"
"    Change INSERT LINE key to vi Open-new-line (O) command:
"
map  ^[L  O
"
"    Change CLEAR DISPLAY key to vi Delete-to-bottom-of-screen (dL) command:
"
map  ^[J  dL
"
"    Change CLEAR LINE key to vi Delete-to-end-of-line (D) command:
"
map  ^[K  D
"
"    Change Left Arrow key (insert mode ONLY) to BACKSPACE:
"
map! ^[D ^H
"
"    Change Right Arrow key (insert mode ONLY) to Move Right:
"
map! ^[C ^V
"
"    Change Up Arrow key (insert mode ONLY) to Move Right:
"
map! ^[A ^V
"
"    Change Down Arrow key (insert mode ONLY) to RETURN:
"
map! ^[B ^M
"
"    Change CTRL-X to "adjust both margins on current paragraph":
"
map ^X {!}adjust -j^M
"
" S = save current vi buffer contents and run spell on it, putting list of
"     misspelled words at the end of the vi buffer.
map    S G:w!^M:r!spell %^M
"
```

The last map command in the file shows how to use the [S] key in command mode to automatically run the *spell* command on the current buffer and append the output of *spell* to the buffer file. Here is how it works:

- When the [S] key is pressed, the **G** command moves the cursor to the last line of the file.
- The **:w!** command then writes the buffer to the original file so that it is up-to-date.
- The **:r!spell %** command tells the editor to run *spell* on the current filename then read the standard output from *spell* into the current buffer after the current (last) line in the file.
- The **^M** characters are carriage-return characters that separate the commands in the mapping.

Datacomm Protocol Conflicts

Most HP-UX systems use DC1/DC3 handshake protocol between the HP-UX computer and user terminals where a DC3 ([CTRL]-[S]) character from the terminal tells the computer to suspend output to the terminal and a DC1 ([CTRL]-[Q]) tells the computer to resume sending output. However, this does not prevent users or other factors from configuring your terminal for ENQ/ACK protocol even though it is not being used, or from setting EOT [CTRL]-[D] as an end-of-file marker character.

When a *vi* (or *ex*) editing session is in progress, the editor takes over all terminal interface functions from the shell so that it can implement special key commands (such as using the **h**, **j**, **k**, and **l** keys for cursor control) and maintain screen displays. Since the terminal interface drivers still handle all characters being transferred, they interpret datacomm protocol characters being used as communication signals and may discard some characters interpreted as datacomm control data even though they are needed by *vi*. Consequently, when DC1/DC3 handshaking is configured but ENQ/ACK is also enabled, any ENQ or ACK characters are lost, thus leading to certain functions such as [CTRL]-[F] being disabled as mentioned in Chapter 3.

Using Ex 13

The *ex* editor, which is essentially an extended version of *ed* is used mainly by those who do not have access to a CRT display terminal. It is much more comprehensive and more versatile than the *edit* version that uses predefined defaults for some options to better fit the needs of beginning and casual users, but is rarely used by most HP-UX users except as accessed from *vi*. In this tutorial, default settings are assumed for all command options unless stated otherwise.

Starting ex

When invoked, *ex* (and *vi*) uses the environment variable TERM to determine the terminal type. If a entry in the *terminfo* data base matches the terminal described by the TERM variable, that description is used. If there is a variable EXINIT in the environment, the editor executes the commands contained in that variable. Otherwise, if there is a file *.exrc* in your HOME directory, *ex* reads commands from that file to configure the editor. Option-setting commands placed in EXINIT or *.exrc* are executed before each editor session. In addition, the editor looks for another *.exrc* file

The *ex* start-up command has the following prototype:

```
ex [-] [-v] [-t <tag>] [-r][-l] [-w<n>] [-x] [-R] [+ <command>] <name> . . .
```

where brackets ([]) surround optional command parameters. The most common case edits a single file with no options, i.e.:

```
ex name
```

Command-line options function as follows:

-	Suppresses all interactive-user feedback; useful when processing editor scripts in command files.
-v	Equivalent to using *vi* rather than *ex*.
-t	Equivalent to an initial tag command. Edits the file containing the tag and positions the editor at its definition.

`-r` Used in recovering after an editor or system crash. retrieves the last saved version of the named file or, if no file is specified, types a list of saved files.

`-l` Sets up for editing LISP, by setting the showmatch and lisp options.

`-w` Sets the default window size to n, and is useful on dial-ups to start in small windows.

`-x` Causes *ex* to prompt for a key that is then used to encrypt and decrypt the contents of the file. The file should have been previously encrypted using the same key, see *crypt*(1).

`-R` Sets the read-only option at the start.

name Indicates which file(s) to edit.

An argument of the form + <command> indicates that the editor should begin by executing the specified command. If <command> is omitted, the argument defaults to "S", initially positioning the editor at the last line of the first file. Other useful commands here are scanning patterns of the form */pattern*, or line numbers such as `+100` (which starts at line 100).

File Manipulation

Current File

In normal use, *ex* is used to edit the contents of a single file whose name is specified by the **current** filename. In a typical editing sequence, the name of the file to be edited becomes the current filename, and the original file contents are copied into a buffer which is actually a temporary buffer file. *Ex* performs all editing actions on the buffer file. Changes made to the buffer have no effect on the file being edited unless and until the original file is replaced by the edited buffer contents (by use of a write command). The write operation destroys the original file and replaces it with the edited version.

The current file is almost always treated as having been edited. This means that the buffer file contents are logically connected with the current file name so that writing the current buffer contents onto that file, even if it exists, is a reasonable action. If the original file has not been edited, then *ex* will not normally write on it if it already exists (a "not edited" message is returned when the write operation is attempted).

Alternate File

Each time the current filename is given a new value, the previous current file name 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

Filenames within the editor can be specified using normal shell-expansion conventions. In addition, the character `%` in filenames is replaced by the current file name; the character `#` is replaced by the alternate file name (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).

Multiple Files and Named Buffers

If the command line specifies more than one file to be edited, the first file is edited as previously explained. Command-line arguments and file names for the first and subsequent files to be edited are placed in the argument list (the current argument list can be displayed by using the *args* command). When you are ready to edit the next file in the list, use the *next* command. If you want to destroy the original argument list and associated file names, replacing them with a new list, append the desired new arguments and file names to the *next* command. HP-UX then expands the *next* command with its new arguments. The resulting list of names becomes the new argument list; the old list is destroyed, and *ex* edits the first file on the new list.

Ex has a group of named buffers that are particularly useful for saving blocks of text during normal editing, especially when editing multiple files. These buffers are similar to the normal buffer file, except that only a limited number of operations can be used with them. The buffers have names *a* or *A* through *z* or *Z*. Uppercase and lowercase names refer to the same buffers, but commands *append to* uppercase-named buffers and *replace* lowercase-named buffers.

Read-only Operation

You can use *ex* in read-only mode to look at files that you have no intention of modifying, thus preventing the possibility of accidentally overwriting a file. Read-only mode is active when the **readonly** option is set by:

- Using the `-R` command-line option,
- The *view* command line invocation, or
- By setting the `readonly` option.

Read-only can be cleared by setting noreadonly (type: `:set noreadonly` **RETURN**). You still can write to a file, even while in read-only mode, by indicating to the editor that you really know what you are doing. This is done by writing to a different file or using the `!` form of the *ex* **write** command, but you must have write permission on the file being overwritten (if the file is marked as read-only, it cannot be overwritten except by super-user).

Exceptional Conditions

Errors and Interrupts

When errors occur, *ex* prints an error diagnostic and, optionally, rings the terminal bell. If the primary input is from a file, editor processing terminates. If an interrrupt signal is received, *ex* prints "Interrupt" and returns to its command level. If the primary input is a file, *ex* exits when an interrupt occurs.

Recovering from Hangups and Crashes

If a hangup signal is received and the buffer has been modified since it was last written out, or if the system crashes, either the editor (in the first case) or the system (after it reboots in the second case) 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 hangup or editor crash. To recover a file, you can use the `-r` option. For example, if you were editing the file *resume*, you should change to the directory you were using when the crash occurred, giving the command:

```
ex -r resume
```

After checking that the retrieved file is indeed intact, you can write it back over the original unedited file. The system normally sends you mail, telling you when a file has been saved after a crash. The command:

```
ex -r
```

prints a list of the files that have been saved for you. (In the case of a hangup, the file does not appear in the list, although it can be recovered.)

Editing Modes

Ex has five distinct operating modes:

- **Command mode** where commands are entered when a colon (:) prompt is present and executed each time a complete line is sent.
- **Text-input mode** where *ex* gathers incoming lines of text and places them in the file. Append, insert, and change commands use text-input mode to alter existing text.

  No prompt is printed when you are in text-input mode. To exit this mode, type a period (.) immediately followed by an end-of-line key (**RETURN**). Command mode then resumes.
- **Open** and **visual** modes enable you to perform local editing operations on text in the file. The modes are accessed by commands having the same name. The *open* command displays text, one line at a time, on any terminal, while the *visual* command switches to *vi* and is designed for CRT terminals that have direct screen cursor-addressing capability so *ex* can use the CRT as a window for file-editing changes.
- **Text insertion** mode operates within *open* and *visual* modes.

These modes are discussed elsewhere throughout this tutorial.

Command Structure

ex commands are described in detail in Chapter 10. Most command names are English words, and initial prefixes of the words are acceptable abbreviations. Ambiguous abbreviations are resolved in favor of the more commonly used commands (for example, the command *substitute* can be abbreviated *s*, while the shortest available abbreviation for *set* is *se*.

Command Parameters

Most commands accept prefix addresses specifying which line(s) they are to affect. The forms these addresses can take is discussed in Chapter 10 (as well as in the *sed* tutorial elsewhere in this volume). Some commands also accept or require a trailing count specifying the number of lines to be affected by the command (if rounding is necessary, the number is rounded down). Thus the command `10p` prints the tenth line in the buffer while `delete 5` deletes five lines from the buffer, starting with the current line.

Some commands require other information or parameters that are always appended following the command name; for example, option names in a **set** command, a file name in an **edit** command, a regular expression in a **substitute** command, or a target address for a **copy** command as in **1,5 copy 25**.

Command Variants

Several *ex* commands have two distinct variants. The variant form of the command is invoked by placing an exclamation point (`!`) immediately after the command name. Some of the default variants can be controlled by options; in this case, the `!` serves to toggle the default.

Flags After Commands

The characters **#**, **p**, and **l** can be placed after many commands (a **p** or **l** must be preceded by a blank or tab except in the single special case **dp**). The commands abbreviated by these three characters are executed after the command completes. Since *ex* normally prints the new current line after each change, **p** is rarely necessary. Any number of + or − characters can also be given with these flags. If they appear, the specified offset is applied to the current line value before the printing command is executed.

Comments

Comment commands are ignored by the editor. This feature is useful when making complex editor scripts (used with the **source** command) where explanatory comments are needed. Any line beginning with a double quotation mark (")is treated as a comment and no action results. Comments beginning with " can also be placed at the ends of commands, except in cases where they could be confused as part of text (as in shell escape sequences or in **substitute** or **map** commands).

Multiple Commands per Line

Multiple commands can be combined on a single line by separating adjacent commands with a | character. However, global commands, comments, and the shell escape **!** must be the last command on a line because they are not terminated by a |.

Reporting Large Changes

Most commands that change the editor buffer contents give feedback whenever the scope of the change exceeds a threshold set by the **report** option (described in Chapter 12). This feedback helps detect undesirably large changes so that they can be quickly and easily reversed with an **undo**. When using commands that have a 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 the threshold.

Additional Topics

The full set of *ex* editor commands are described in Chapter 10. Chapter 12 details the use of the **set** command which is used to alter various editor configuration parameters. Refer to those chapters and other related topics elsewhere for information about how to use commands. This chapter has been restricted to a general discussion of how the use the *ex* editor program in contrast with the *vi* editor.

Index

a

b

c

d

e

f

g

h

i

j

l

m

n

o

p

q

r

S

t

u

v

w

y

Table of Contents

sed: A Non-Interactive Streaming Editor 1

Introduction

If you have been around HP-UX or other UNIX-based systems, you have no doubt heard that *sed* is a powerful text editor, but few people, even in rather large corporations with a large number of installed systems, know more than a minimal amount about the program and its capabilities. This tutorial is the result of in-depth examination of validation tests and other tools used in maintaining the integrity of the program in order to discover its deep, dark secrets as well as extensive testing of the examples used. So, for probably the first time in history, *sed* is being brought out of obscurity and into full public view.

sed is a high-speed, non-interactive, streaming (batch process) text editor. It is most commonly used for complex editing operations on large files or repetitive operations on a large number of files where interactive editing with an editor such as *vi* is too tedious or time consuming. Editing operations to be performed are defined by the editor commands which can be included as part of the HP-UX *sed* command line or stored in a separate commands file.

The commands, which can range from very simple commands included in the HP-UX *sed* command line to extremely complex command streams stored in separate files, specify what alterations are to be performed on input text as a function of current text contents. For long and complex operations on many large files, *sed* can be run as a background process, freeing your terminal for other tasks while it runs. However, don't be surprised if it finishes much sooner than expected, because *sed* is a very fast editor.

While using *sed* is conceptually rather simple, many users have experienced difficulty with existing learning aids to date because most writings tend to be terse and difficult to understand or are targeted at the beginner and do not address more advanced topics. Some users have treated *sed* as an unpleasant task that must be learned instead of a powerful tool that can often make tedious work a breeze. This tutorial has been structured to eliminate your fears of the unknown by making the unknown knowable, and not only knowable but also usefully simple. Many examples have been included after testing them to be sure they actually work the way they are explained.

Manual Organization

This tutorial is divided into three chapters:

- Chapter 1 introduces the *sed* editor and explains how the editor operates and how to start it using the HP-UX *sed* command.
- Chapter 2 explains how editor commands are constructed. It discusses addressing, commands, and related arguments, including typical examples.
- Chapter 3 discusses command script files and how to arrange commands in the correct sequence to obtain desired results. Non-trivial examples are provided to support the concepts that are covered.

Editor Operation

sed is a very fast editor because it does not maintain large blocks of text in memory. Instead, text is read from the input file, one line at a time. As each line is read into a location in memory called the **pattern space**, the command(s) contained in the editor command script file (or the command provided on the HP-UX command line when the editor was started) are applied to the line, indicated changes are made if any, and the result is written to standard output before the next line is read into the pattern space. This "bucket brigade" approach greatly reduces computing resources and can reduce many weeks of tedious interactive editing to a very few minutes of high-speed processing when the tasks to be performed are well-defined and easy to reduce to sequences that resemble a computer program in their order of execution.

By eliminating the use of temporary files and holding only a few lines in memory at any given time, the maximum file size that *sed* can handle is limited only by the amount of external file storage space available for the file being edited and the resulting output file. By supporting command scripts stored in existing files, *sed* can run faster than virtually any known editor, even when that editor can be driven from comparable scripts. The availability of command script support greatly improves operating reliability through less need for repetitive typing and the attending errors, although the speed and efficiency comes at the price of less interaction, and thus no ability to verify that the change made is really what was wanted or intended.

Commands

You need to clearly understand two categories of commands when using this manual:

- The HP-UX *sed* command and its options which are used to run the editor program.
- Editor commands that are interpreted by the *sed* program and which specify what editing actions are to be performed by the program as it processes the incoming input text file(s).

Sed Command

The HP-UX *sed* command is interpreted by the HP-UX user shell and can include the following arguments in the command line when the command is typed on the terminal keyboard:

- A **−f** option with a filename argument specifying a script file that contains a group of one or more editor commands that can be interpreted and executed by the *sed* editor, or a **−e** option and an argument consisting of a valid *sed* editor command included in the HP-UX command line.
- A list of one or more source files to be edited. If multiple input files are specified in a single command, they are concatentated into a single output file.
- Output redirection or other programming device to send editor output to a location other than standard output.

These items are discussed in greater detail in the remainder of this chapter.

Editor Commands

Editor commands (such as append, substitute, insert, print, etc.) are interpreted and executed by the editor as it processes the input file and produces a new output file. These commands must be included in the HP-UX command line following the **−e** option or be placed in sequence in an external commands stream file whose name is specifed as an argument to the **−f** option in the HP-UX *sed* command line.

Editor commands are discussed at length in the next chapter.

Invoking the HP-UX Sed Command

The recognized format for the HP-UX *sed* command varies, depending on whether the editor command(s) are included as arguments in the HP-UX command line or are placed in a script file, and whether default output is to be produced or suppressed. All editor output is sent to standard output unless specified otherwise by a redirection directive or other programming device. Command format for each HP-UX command line option is as follows:

Edit File Using Commands Contained in a Commands Script:

Using an **−f** option in the HP-UX *sed* command line indicates that the editor commands are to be taken from an editor script file while processing the source file *file*:

sed −f *script_file file*

script_file is a file containing a stream of valid *sed* commands. *file* is the file to be edited. If *file* is not specified, input is taken from standard input. Multiple files can also be specified, in which case they are concatenated together and sent to standard output after editing. This command form is most commonly used for multiple commands and complex operations.

Edit File Using Commands Provided in HP-UX Command Line:

An **−e** option in the HP-UX *sed* command line indicates that all editor commands are present as an argument on the HP-UX command line for processing the source file *file*:

sed −e '⟨*editor_command*⟩' *file*

If *file* is not included in command line, input is taken from standard input. Multiple files can also be specified, in which case they are concatenated together and sent to standard output after editing. Note the use of single quotes (') around the *sed* command to protect it from interpretation by the HP-UX command shell. This command form is most commonly used for one-line commands and simple operations.

A variation on this form can be used to simulate a script by including multiple **−e** options on a single command line. This method is discussed in detail in Chapter 3.

Note

A blank (space or tab) is **required** after the **−e** option before the corresponding command and after the **−f** option before the stream file name.

Edit File but Suppress Default Output:

The **−n** option can be used with either the **−f** or **−e** option to suppress output on all except for those lines identified by a print command or flag in the *sed* command or editor command script. The **−n** option is used in either of the following two forms:

sed −n −f *script_file file*

or

sed −n −e '⟨*editor_command*⟩' *file*

The **−n** in the HP-UX command line tells *sed* to suppress all output to standard output except for the following situations:

- Current line in pattern space matches the address field of an editor print command for single-line pattern spaces (**p** command) or for multi-line pattern spaces (**P** command) as described later in this tutorial.
- Current line matches a substitute command that includes a print flag (**p**) after the ⟨*replacement_text*⟩ argument in the substitute command line.

Note

Unlike many HP-UX commands, the **−n** option cannot be combined with the **−e** or **−f** as a single argument (as in **−ne** or **−nf**). The **−n**, **−e**, and **f** options must be separate arguments as shown above whenever the **−n** option is present in the command line.

Program Start-up and Operation

When HP-UX executes the *sed* command line, it creates a new process and starts the editor program. *sed* then opens the stream file containing the editor commands if the **−f** option is specified or accesses the editor command(s) included in the HP-UX command line if the **−e** option is specified.

The editor commands are then compiled into a form that will be moderately efficient during execution (comments are removed for better speed, for example) when the commands are actually applied to the input file. Commands are compiled in the order in which they are encountered which is also the general order in which they will be attempted at execution time. Commands are applied, one at a time, to the pattern space text in its current form. That means that the pattern space always contains the result from the most recently executed *sed* editor command that affected the pattern space contents.

The order in which commands are applied to the pattern space can be modified by using the two branching commands: **t** (for conditional branching), and **b** (for unconditional branching). Labels are used to identify destinations when branching commands are used.

Commands can also be grouped under a single address or address range for multiple operations on a given line or set of lines by using the grouping symbols **{** and **}**.

After the commands have been compiled, the text file(s) to be edited is then opened (one file at a time if more than one file is specified), and editing proceeds according to the commands provided.

Each line in the input text file(s) is read into the pattern space, one line at a time. The editor commands specified in the HP-UX *sed* command or in the commands stream file are applied in sequence to each line while it is in the pattern space, and the results are sent to standard output or elsewhere as specified in the HP-UX command option or by the individual editor commands themselves. When the last line has been read from the input text file(s) and the last applicable command has been executed on that line, the program closes all files and terminates.

An alternate **quit** editor command can be used to terminate the edit on a given line as determined by the corresponding address if you want to edit only part of the file. Lines in the input file that follow the address of the line matching the quit command are omitted from the output. Judicious use of grouping and quit commands can provide an easy means for extracting an edited form of only part of an existing file without manually going back later and deleting unwanted text from the output file.

Most of the remainder of this tutorial discusses how to construct and use editor commands and scripts. Examples are provided to illustrate typical use.

Sed Editor Command Script Limits

Certain limits are imposed on *sed* users by the program itself. These limits are discussed in greater detail elsewhere (mainly in Chapter 3), but are collected here for easy reference:

- Up to 10 files can be open for writing or reading in conjunction with **w** and **r** commands and **w** flags (see Chapter 2) in addition to the input file and commands stream file specified in the HP-UX command line.
- As many as 100 editor commands can be included in a commands stream file in addition to labels and comments. An error is generated if this limit is exceeded. This limitation is listed as a bug on the *sed*(1) manual page in the *HP-UX Reference.* Multiple commands that are collected within braces for grouping to a common address are still treated as separate commands when determining compliance with the 100-command limit.

  The number of commands that can be included in an HP-UX *sed* command line as arguments to one or more **−e** options is limited by the number of characters allowed in the HP-UX command-line buffer and other factors such as the number of arguments allowed by the shell commands interpreter as well as the *sed* program itself. It is difficult to determine the number of **−e** options that can be used in a particular situation, but it is safe to say that it is usually much less than 100 commands.
- Up to 50 labels can be included in the commands stream for conditional and unconditional branching. An error is generated if this limit is exceeded.
- Label names can contain as many as seven alphanumeric characters. Eight or more characters produce an error message.
- There is no limit on the number of comment lines that can be included in the commands stream file. Scripts are compiled at the beginning of an edit, and comments are removed from the compiled commands at that time. Thus, comments do not require significant additional processing time and do not slow down complex edits on large files. Their use makes debugging and interpretation of the script months (or even a few hours) later much easier, so their use is strongly recommended.
- Grouped commands ({ and }) can be nested up to 100 levels deep. It's not obvious why one would need that many nesting levels with only 100 commands per script available, but that's the rule.

Forming Editor Commands 2

This chapter discusses how to construct *sed* editor commands. It begins with a brief overview of what components are used to build a command, then explains each component part in greater detail. Commands are grouped into classifications according to use and each is explained thoroughly. A solid understanding of the basic concepts introduced in this chapter is necessary before you can effectively use *sed* commands, and it can be gained in a couple of hours or so by most users who already have some experience in using HP-UX and regular expressions. If your understanding of regular expressions is weak, you will need to spend some time in the tutorial on regular expressions earlier in this volume. However, you do not need to understand them thoroughly before you can benefit from reading this chapter. As you progress, you may find it beneficial to switch back and forth between tutorials as questions arise.

Be Careful

If you are a frequent *ed* editor user, be aware that many of the commands and regular expression constuctions in *sed* closely resemble similar commands and expressions in the *ed* editor, partly because *sed* is descended from *ed*. However, casually using *ed* commands in a *sed* environment can produce results very unlike what a habitual *ed* user might expect, much to his or her chagrin.

Editor Command Format

All *sed* editor commands have the following general form (square brackets identify the various parts of a command line, but they are not used when constructing an actual *sed* editor command):

[*address1,address2*][*editor_command*][*arguments*]

This general form is used in basically three different variations:

- When no address is specified, the command is applied to every line in the input file:

  [*editor_command*][*arguments*]

- When only one address is specified, the command is applied to every line in the file that is specified by *address*:

  [*single_address*][*editor_command*][*arguments*]

- When two addresses are specified, the command is applied to every line in the input file starting with the first line identified by *address1*, and continues through the first subsequent line that is indentified by *address2*.

  [*address1,address2*][*editor_command*][*arguments*]

Command components are as follows:

address1,address2 — Zero, one, or two addresses that identify lines in the file that are to be affected by the operations or changes specified by *editor_command* and *arguments*. Line addresses can be specified by using line numbers, or regular expressions can be used to locate text within a line. When regular expressions are used, the address specifies any line containing one or more text patterns that match the pattern specified by the regular expression (for example, any line containing a character pattern such as *xyz* or some other pattern as defined by a syntactically correct regular expression).

editor_command — Specifies the type of operation to be performed on the lines identified by the address(es) if present. If no address is present in the command line, *editor_command* is applied to every line in the file. Editor commands fall into several general categories that include full-line commands (such as delete line, append lines after current line, insert lines before current line, and change current line to specified new text), the *substitute* command that replaces all or part of the existing text within a specified line with new text, input/output commands, and others.

arguments — Replacement text, flags, pattern-recognition expressions, and other elements related to the specific *editor_command* contained on a given editor command line. Arguments vary widely depending on the command being used, as discussed later in conjunction with each command.

Whitespace in the form of one or more blanks and/or tabs is allowed between addresses and between the address field and the *editor_command* field in each command line. Whitespace between the editor command and its arguments may or may not be allowed, depending on the editor command and the context in which it is used.

Pattern Space

Before discussing how to build addresses and commands using regular expressions, it is very important that you clearly understand the use of the pattern space which holds the line or lines being processed at any given moment.

In normal operation, whenever a line is processed and sent to output, a new line is read into the pattern space from the input file to replace it. After editing, that line is sent to output and a new line replaces it in turn, and so forth.

However, there are situations where the line in the current pattern space might be changed into multiple lines. For example, substituting one or more patterns with replacement text that includes end-of-line characters can split the line into two or more lines (depending on the desired result). In other situations, the **D**, **P**, and **N** commands can be used to append new lines at the end of the current pattern space, delete the first part of the current pattern space (up to the first newline – also known as end-of-line), or print the same to standard output. This means that the current pattern space can contain what will become several lines in the output from the editor. When multiple input or output lines reside in the pattern space, they are separated by what is called **embedded newlines**. Embedded newlines is nothing more than a fancy way of saying that multiple input or output lines are being treated as a single line for processing purposes.

Multiple lines residing together in the pattern space are usually fairly harmless, but they can create some frustrating moments when you are trying to understand why a command script doesn't work correctly. Most problems arise because regular expressions treat the entire pattern space as a single line with embedded newlines while most users prefer to think of the pattern space as containing multiple lines. As usual, when such disputes arise the computer always wins – just like when you try to fight with a credit card company's computer over a billing error. So remember:

Note

When using regular expressions in addresses and substitution commands, remember that the entire pattern space is treated as a single line with embedded newlines that are part of that line even though they serve as end-of-line on each line when they appear in the output. This means that the first character tested in an address or substitution search to match the corresponding regular expression in a line is the first character in the first line in the pattern space. The last character in an address or substitution search is the last character in the last line in the pattern space. The regular expression special characters ^ and **$** represent the beginning and end of the current pattern space, respectively, and have no relationship to individual embedded newlines within the pattern space.

The remainder of this chapter explains how to build each component of an editor command line into a usable command for inclusion as an argument to an HP-UX *sed* command using the **–e** option or in a *sed* command script (stream) file using the **–f** option.

Constructing Line Addresses

Line addresses appear on each editor command line; zero, one, or two addresses per line. Each address, if present, can have one of two general forms; a numerical line-number address, or a context address which consists of a regular expression used to identify one or more specific lines in the file based on textual content within the line (thus the name, context address).

Line Address Defined

In this discussion, the term *line address* refers to either of two possible address forms:

- **Numerical addresses**: A line number used as an address refers to the line numbering sequence in the input file.
- **Context addresses**: A regular expression used as an address refers to any line in the entire contents of the input file that contains a pattern that matches the specified address while it is being processed in the current pattern space. If the line has been subjected to alteration by previous commands while residing in the pattern space, the address of the current command must match the line in its **current state**, which may be very similar to the original text line that was placed in the pattern space when it was read from the input file, but it could be very different, depending on what changes have already been made to the line by previous commands in the commands script.

Interpreting Line Addresses

Line number

Any valid line number in the value range of **1** through the number of the last line in the input file can be used for either or both addresses. When two addresses are specified for starting and ending line numbers, they must be separated by a comma (,). For example, the address pair **2,24** indicates that the editor command is to be applied to all lines starting with line 2 and continuing through line 24. The value of the second address must be greater than the first. The dollar currency symbol (**$**) can be used as a line address to represent the last line in a file when the line number is not known or subject to change. Thus the address **13,$** specifies all lines from line 13 through the last line in the file. Obviously, the **$** must always occupy the second position in a two-address field.

Regular Expression (Context Address)

It is often preferable to identify a line by its contents rather than by line number, especially since most text files and many higher level computer programs do not use line numbers. This is easily accomplished by replacing the numerical address with a regular expression between slash characters (*/regular_expression/*). For example, the address pair:

/start text block A/,/end text block A/

affects all lines starting with the line containing the text pattern **start text block A** and ending with the subsequent line **end text block A**. Other examples of this technique are demonstrated in later topics.

Line addresses can appear in any of the following combinations on any given editor command line:

- No address implies that the editor command on that line is to be applied equally to all lines in the input file.
- A single address implies that the editor command is to be applied to all lines that match the specified address. If the address is in the form of a number, *n*, only line *n* in the file is to be subjected to the command. If the address is in the form of a text pattern or other regular expression, any line in the input file that contains a pattern that matches the address expression is to be subjected to the command.
- A double address of the form *address1,address2* indicates that the command is to be applied to all lines starting with the line identified by *address1* and continuing through the line identified by *address2* where *address1* and/or *address2* can be a line number or a regular expression.

When a command is preceded by two addresses, the double address is interpreted as follows:

- If one or both line addresses reference line numbers as indicated by their numerical or partially numerical form (*n1,n2*; *n1,expression2*; or *expression1,n2*;) the command is applied as follows:
  - ***n1,n2:*** All lines starting with line *n1* through line *n2* are processed by *editor_command.* Lines outside the specified range are ignored. If *n2* is less than *n1*, the command is ignored and no error is generated.
  - ***n1,expression2:*** All lines starting with line *n1* through the first subsequent line containing a text pattern that matches the regular expression *expression2* are processed by *editor_command.* Lines outside the specified range are ignored.
  - ***expression1,n2:*** All lines starting with the first line containing a text pattern that matches the regular expression *expression1* through line *n2* are processed by *editor_command.* Lines outside the specified range are ignored. If the line that matches address *expression1* occurs later in the file than line *n2*, all lines from the line containing *expression1* through end of file are subjected to the corresponding command.

- If both addresses are regular expressions meaning that the address has the form *expression1,expression2*, all lines starting with the first line that contains *expression1* through the next subsequent line containing *expression2* are processed by the editor command specified on the same command line. If another line containing *expression1* appears in the file after the line containing *expression2*, processing by *editor_command* resumes on that line and again continues until a line containing *expression2* (or end-of-file) is again encountered.

  If only one line in the file contains *expression1* and one other line contains *expression2* but the line containing the second expression precedes the line containing the *expression1*, the command is applied to the entire remaining file starting at the line containing *expression1*. Two-address operation is equivalent to setting a flag to start command execution on all lines when *expression1* is matched to a line, then leaving the flag set until *expression2* is matched or end-of-file is reached, whichever occurs first.

Note

When two addresses are used with an editor command and the second address is a regular expression, *sed* does not search the line identified by the first address to determine whether the second address also exists in that same line. Thus it is impossible to restrict editor commands to a single line when a regular expression is used for the second address.

If the second address is a line number, the operation always terminates on that line, even if the first address (whether line number or regular expression) refers to the same line.

Constructing Context Addresses

The use of regular expressions is discussed at length earlier in this volume in the tutorial on regular expressions. The following table is a short summary of regular expressions and their use in the *sed* environment for your convenience. Examples in later topics show how to use regular expressions for addressing and in substitutions.

Regular Expression Character Interpretation

Character	Matching Text
^	Matches an imaginary zero-width character at the beginning of the pattern space if it is the first character in the expression (in any other position, it is treated as a normal character). Often used to match a string of text that starts at the beginning of the pattern space without matching any identical strings occuring elsewhere.
$	Matches an imaginary zero-width character at the end of the pattern space. Often used to match a string of text that terminates at the end of the pattern space without matching an identical string occuring elsewhere.
\n	Matches a newline (end-of-line) character placed within the pattern space instead of at its normal end-of-line position. Does not match a newline character at the end of the pattern space.
.	A single-character regular expression that represents any arbitrary character except newline (end-of-line). Frequently used with * to represent an arbitrary block of text within a line between two other text patterns or before/after a text pattern. .. represents two adjacent arbitrary characters. The form .* cannot be used to span across one or more embedded newlines.
*	Matches zero or more of the immediate preceding one-character regular expression. Thus **xy*z** matches **xz**, **xyz**, **xyyz** and **xyyyyyyz**. Likewise, **ab.*xyz** matches **ab** followed by **xyz** with any amount of arbitrary text (except newline) between the two.
[*chars*]	Text pattern character can match any one character in the group between the [and] provided the first character is not a circumflex (^). If the first character is a circumflex, the match is tested against all characters **except** the characters between the [and]. To include circumflex in the group of matching characters, place it in any other position. To match [or], place them in the first position(s) in the group as in [[]**abc**^].

Regular Expression Character Interpretation (continued)

Character	Matching Text
\(*string* \ )	The delimiters \(and \) are used to isolate part of a larger regular expression for use in substitution and addressing operations. *string* is an ordinary regular expression of one or more characters. Up to nine sub-expressions can be defined by this method. Patterns in a line that match a given sub-expression can be specified in related operations by using the *digit* specifier described next.
digit	Where *digit* is a single-digit numerical value. Specifies the text pattern that matches the *d*th sub-expression in the preceding regular expression on the same line. For example, **\2** represents the text that matches expression "the" in the regular expression **\(over\) \(the\) \(moon\)**. See examples later in this chapter for more information.
any other	Ordinary typing characters except those above are one-character regular expressions that match only themselves in a pattern search.

Note

Be careful when searching for patterns that match regular expressions because matches may not produce the desired result if the expression is incorrectly structured. For example, to isolate the word "**the**" requires three (or four) pattern searches: (1) " the " preceded and followed by a space, (2) "the" preceded by beginning of pattern space character (^) and followed by a space, and (3 and 4) an expression to identify "the" (preceded by a space) preceding a newline and/or end of pattern space.

Using Sub-Expressions in Addresses

In the preceding table of regular expressions, use of the character pairs \(and \) as delimiters to isolate part or parts of a regular expression as an aid in substitutions and addressing was discussed. This obscure and often confusing feature is a very useful mechanism for identifying certain types of patterns in text files such as lines that contain repeating words or patterns, the same word twice in a row such as here here, and others, although its usefulness is not limited to these cases.

An example from the *HP-UX Reference* manual entry for *ed*(1) says you can find a line consisting of two repeated appearances of the same string by using the address **^\(.*\)\1$**. From this example, it is tempting to try a similar addressing expression to identify lines containing a single word twice in succession in a sentence like like this this. The most obvious expression to attempt would be a very simple address of the following form where the white space between words is assumed to consist of one or more spaces and/or tabs (in other words, arbitrary white space except beginning or end of line):

/ \(. * \) [*space tab*] [*space tab*] * \ 1 /

But when you discover that the address matches every line in the file except blank lines, you begin to scratch your head in bewilderment. Condolences are in order. You are among the many victims of the asterisk which says **zero** or more of the previous single-character regular expression which, in this case, is the . representing anything except newline. Thus the address matches any line containing one or more spaces and/or tabs preceded and followed by zero or more other characters of any kind except newline which reduces to any line containing one or more spaces and/or tabs.

You can add a . before each .* to specify that at least one other arbitrary character is required. This addition helps, but now the address matches lines containing "this system", "when new", "command and", and even lines starting with three or more spaces or tabs; still not what we want.

After careful evaluation, it can be readily determined that successive patterns such as improperly repeated words can occur **at** beginning of line or **after** beginning of line (which includes end-of-line). Thus two addresses are needed. The first case is addressed this way:

/ ^ \([^ *space tab*] . . * \) [*space tab*] [*space tab*] * \ 1 /

and the second address is:

/ [*space tab*] \([^ *space tab*] . . * \) [*space tab*] [*space tab*] * \ 1 /

In both addresses, there are no blanks except where indicated by *space* or *tab.* The two addresses are interpreted as follows: At beginning of line or starting at any blank (space or tab), look for a non-blank character followed by one or more arbitrary characters in turn followed by another one or more blanks and the same non-blank text pattern as before.

If you know that there are no tabs in the text block, replace the expression [*space tab*] with a space, thus producing the addresses:

/ ^ \(. . * \) *space space* * \ 1 /

and

/ *space* \(. . * \) *space space* * \ 1 /

Other examples of using \(and \) with *digit* are included later in this chapter under the substitute command.

Editor Commands

sed editor commands are grouped into the following general categories:

Whole-Line Commands

These commands include:

- **delete** lines,
- Read **next** line,
- **append** new text lines after current line,
- **insert** new text lines before current line, and
- **change** addressed lines to new text.

They operate only in conjunction with full lines of text; one or more full lines in the input file, and one or more full lines of replacement text in insert, replace, and change operations.

Substitute Command

The **substitute** command changes part of an addressed line by conducting a context search within the line and altering the line accordingly.

Transform Command

Related to the substitute command, the **transform** command is used to singly replace certain characters whenever they occur in an addressed line with another character chosen from a corresponding set of replacement characters.

Input/Output Commands

These commands include:

- **print** addressed lines on standard output,
- **write** addressed lines to specified ⟨filename⟩ (often used to keep a record of changed lines for future reference), and
- **read** contents of specified ⟨filename⟩ and append on standard output after addressed line (merge text from external file).

Commands to Process Multiple Input Lines Simultaneously

These commands are used to combine multiple input lines into a single pattern space then process the entire pattern space. They include:

- append **Next** line into pattern space,
- **Delete** up to first newline character in pattern space, and
- **Print** up to and including the first newline (entire first line) in pattern space (send to standard output).

Hold and Get Commands

These commands manipulate text between the current pattern space and a separate hold pattern space (buffer). They include:

- Copy pattern space into **hold** area,
- Append pattern space to current **Hold** pattern,
- **get** contents of hold space and place in pattern space,
- **Get** contents of hold space and append to current pattern space contents, and
- **eXchange** contents of the pattern space and hold area.

Flow-of-Control Commands

These commands do not alter input lines, but serve, rather, as a means for controlling the application of editor commands to lines selected by the address(es) associated with the flow-control commands. They include:

- Address inversion (operate on all addresses **except** those matching the command address),
- Command grouping for multiple operations on a single line or pattern space,
- Labelling and branching to labels, and
- Testing for successful substitutions in the current pattern space.

Miscellaneous Commands

There are two commands in this group:

- Print line number (=) matched by address on standard output, and
- **quit**.

Constructing Editor Commands

As mentioned earlier, *sed* commands have the following general forms:

- For commands that are to be applied to all lines in the input file, use:

  command [*optional_arguments*]

- For commands applied to a single line or lines containing a given text pattern, use:

  address command [*optional_arguments*]

- For commands that affect a group of contiguous lines starting with *address1* and continuing through *address2*, use:

  address1, *address2 command* [*optional_arguments*]

  Which of the three basic forms you select depends on the command and what needs to be accomplished. This section explains how to construct the commands, determine what address forms to use, and decide how to arrange command arguments for the correct result. Commands are presented in groups in the same sequence as they are listed in the previous section.

Whole-Line Commands

These commands operate on an entire line or on a group of complete lines. They include delete or change line(s), insert/append text before/after line(s), and read next line in input file.

Whole-Line Commands

Command	No. of Addresses	
d (delete line(s))	0, 1, or 2	Delete all lines matched by the address(es) specified with the command; do not write them to the output. When a line is deleted, the pattern space is empty and no further commands are attempted on the corpse. A new line is read from the input file into the pattern space. The list of editing commands is then restarted from the beginning on the new line.
n (next line)	0, 1, or 2	Reads the next line from the input file, replacing the current line in the pattern space. The current line is written to the output if it should be before being replaced by the new line. Execution of editing commands continues with the command following the **n** command instead of starting over at the beginning of the commands (unless the **n** command happens to be the last command in the list).
a ⟨text⟩ (append lines)	0 or 1	This command appends ⟨text⟩ after the addressed line in standard output. The **a** command (together with its address, if any, must be on a line by itself followed by one or more lines of ⟨text⟩. If ⟨text⟩ is two or more lines, each line except the last must end with a backslash escape character (\) indicating that the next line is text, and is not to be interpreted as a new command. The first line in ⟨text⟩ that ends with a normal end-of-line terminates the append operation. Once an **a** command is successfully completed, the resulting ⟨text⟩ is sent to standard output at the appropriate time regardless of any subsequent editor commands applied to the line that triggered the append. This means that even if the triggering line is deleted or destroyed, the appended text is still written to standard output. Likewise, if the line (or lines) is altered by a substitute or change command, the altered line(s) is sent to standard output before the appended ⟨text⟩. ⟨text⟩ is not part of the input file, so it is not scanned for address matches and it is completely unaffected by any editing commands. It also does not affect the input file line address counter.

(continued next page)

Whole-Line Commands (continued)

Command	No. of Addresses	
i ⟨text⟩ (insert lines)	0 or 1	Exactly identical to **a** except that ⟨text⟩ is written to standard output **before** the addressed line instead of after it.
c ⟨text⟩ (change lines)	0, 1, or 2	Identical to **i** and **a** except that ⟨text⟩ replaces the addressed line(s) instead of being inserted or appended. Any lines that match the specified address(es) in the change command are deleted and no further editing operations are performed on them. If **c** is preceded by two addresses, all lines in the addressed group of lines are deleted and replaced by a single copy of ⟨text⟩ in the standard output (*sed* does not produce one copy of ⟨text⟩ per deleted line when two addresses are used). If **c** is preceded by one address, each line that matches the address is deleted and replaced by a copy of ⟨text⟩ in the standard output (thus producing a copy of ⟨text⟩ per deleted line when one address is used). If no address is provided, every line in the file is deleted and the output consists of a complete copy of ⟨text⟩ for each line in the file (doesn't sound very useful, but that's the way it works). If an append (**a**) or read file (**r**) command is executed for the same address as the **c** command, ⟨text⟩ from the **c** command is sent to standard output before the text from the **a** or **r** command. If both an **a** and **r** command are associated with the same address as the change, they are written on standard output in the same order as the append and read commands appear in the editor commands sequence. Upon completion of the change and any append/read operations associated with the same address, the next line is read from the input file and execution of editor commands starts over with the first command in the list. The input-file line-address counter is incremented as part of the next-line read operation.

Whole-Line Commands: Important Details

- No whitespace (space or tab) is allowed between the command name and any preceding address(es). If any space is present, a **`command garbled`** error message is produced.
- If two addresses are used and the second address is a regular expression, *sed* does not search the line matching the first address to see if it contains a text pattern matching the second address. Thus, if the line identified by the first address contains a text pattern matching the second address, the command will always be applied to more than one line (to end-of-file if the second-address pattern cannot be matched in a subsequent line) unless the line matching the first address is also the last line in the file.

  On the other hand, if the first address is a regular expression and the second address is a line number, if the first address matches the same line as the second address, only one line is affected by the command.

Using Whole-Line Commands

Here are some typical examples of whole-line commands showing how they are constructed and what they do.

Command Used	Action Taken
3,5d	Delete input file lines 3 through 5.
/^\.mc/d	Delete all lines that begin with **.mc**. This is useful for removing change marks from files that have been marked by the HP-UX command, *diffmk*.
a **This text is appended after** **the addressed line(s).**	Append the specified two lines of new text after every line in the file.
/xyz/a **This text is appended after** **the addressed line(s).**	Append the specified two lines of new text after every line in the file that contains the character string **xyz**.
15i **This text is appended after** **the addressed line(s).**	Insert the specified two lines of new text before line 15 in the input file.
3,/the/c **This text replaces** **the addressed lines.**	Change line 3 through the first subsequent line containing "**the**" to two lines of new text, even if the pattern occurs within a word such as "whe**the**r".

Substitute Command

The **substitute** (and the related **transform** command that is discussed in the next topic) is the only *sed* command that can be used to arbitrarily alter text within an addressed line. Substitute command format is as follows:

- The no-address form performs the substitution on every line in the file that contains a text pattern matching ⟨regular_expression⟩:

  s/⟨*regular_expression*⟩/⟨*replacement_text*⟩/⟨*flags*⟩

- The single-address form performs the substitution on all lines that match *address* and contain a text pattern that matches ⟨regular_expression⟩:

  *address*s/⟨*regular_expression*⟩/⟨*replacement_text*⟩/⟨*flags*⟩

- The two-address form performs the substitution on each line from *address1* through *address2* that contains a text pattern matching ⟨regular_expression⟩:

  address1,*address2*s/⟨*regular_expression*⟩/⟨*replacement_text*⟩/⟨*flags*⟩

Substitute Command: Regular Expression

The text pattern ⟨*regular_expression*⟩ can be delimited by the slash character (/) as shown, or you can use any other character except space and newline. Any valid expression can be used. To use special characters as normal characters, they must be preceded by a backslash (\). Whatever delimiter you use must appear three times in the *arguments* part of the command line: before the regular expression, between the regular expression and the replacement text, and after the replacement text.

Substitute Command: Replacement Text

Replacement text can be arbitrary text. Special characters used in regular expressions and elsewhere have no significance, so they do not need to be preceded by a backslash (\). To include a backslash in the output text, it must be preceded by another backslash; that is, use \\ to get \. If the replacement text consists of more than one line, each line must end with a backslash to protect the end-of-line from interpretation by *sed* as end-of-command-line.

Substitute Command: Delimiters between Regular Expression and Replacement Text

The delimiter (usually a slash, but it can be anything except space or newline) must appear **exactly three** times in the editor command line; namely, before the regular expression, between the expression and the replacement text, and immediately after the replacement text (unless it is preceded by a backslash escape character when used for pattern matching or in regular text). If you need to use the delimiter character in a regular expression or in the replacement text, precede it with a backslash to force *sed* to treat it as a normal text character instead of interpreting it as a representation of something else.

Substitute Command: Special Characters Used in Replacement Text

With the exception of **&** and *digit*, when typing the ⟨*replacement_text*⟩ part of the editor command you can use regular expression special characters ^, **$**, [,], ., and * without preceding them with a backslash escape. The only restrictions you must remember are escaping the delimiter character by preceding it with a backslash, using a backslash before end-of-line if the replacement text has more than one line, and terminating the last line in the replacement text with the delimiter character (usually a slash) followed by any optional flags that might be needed. All other characters in the command line are treated literally without interpretation of any kind. However, if you get confused and want to make sure nothing is misinterpreted, you can always use the \ escape. If it is present but not needed by the command, it is discarded by the editor before making the text substitution.

Substitute Command: Using Matched Text in Replacement Text

There are two types of matched text that can be used in the replacement expression without retyping it:

&	**&** represents that string of text in the current pattern space that matches the complete search expression immediately following the **s** command on the current command line.
digit	*digit* represents the string of text in the current pattern space that matches the *digit*nth sub-expression in the search expression as determined by counting the number of \(sequences in the regular expression that precede the sub-expression. This method of defining and using sub-expressions is explained in greater detail including some examples later in this substitute command section of this chapter.

Thus, to change all occurrences of **filename** to **filename.c**, use the command:

s/filename/&.c/

Substitute Command: Flags

The substitute command accepts three independent flags as optional arguments in the editor command line after the regular expression and replacement text part of the command. These flags can be used in any combination; alone, in pairs, or all three together as your needs might dictate. The recognized flags are:

Substitute Command Flags

Flag	Function
g	If the regular expression matches text in the input line in more than one location, the substitution is performed on every match in the line. If this flag is absent, the substitution occurs only on the first encountered match in the line. Overlapping occurrences of *regular_expression* are ignored and no change is made to either occurrence.
n	Substitute only the **n**th occurrence of *regular_expression* where **n** can have any value from 1 through 512.
p	If a pattern matching the regular expression was found and the substitution made, the line is printed (written) to standard output. If no substitution is made, and the **−n** option was included in the HP-UX *sed* command line, no output is produced meaning that only changed lines appear on the output unless dictated otherwise by another command in the commands script dictates otherwise). Likewise, if other substitution commands in the script include a **p** flag, match the current line, and alter the line, multiple, dissimilar copies of the line are produced, one for each successful substitution (again assuming that the **-n** option was included in the original HP-UX *sed* command line).
w ⟨*filename*⟩	Equivalent to the **p** flag, except that output is written to the specified file instead of standard output. If the file exists before *sed* is run, the file is overwritten. If not, a new file is created with the specified name. As with the **p** flag, multiple dissimilar copies of the line can be written as a result of multiple substitutions on a given line. A single space (not tab) must separate the **w** flag from the filename. Good command writing practice places this flag last, after the command and all other arguments and flags. Up to 10 different files can be open for writing during a *sed* session. If the **read** file command is present in the editor commands stream, only nine different filenames can be specified by various **w** flags and **w** (write) commands.

Substitute Command: Important Details

Remember that the regular expressions used in addresses associated with the **s** command are completely independent from the regular expression used in pattern substitution. This independent relationship proves rather difficult for many beginners. Some of the examples that follow in this section should help clarify the matter considerably.

Using the Substitute Command

Here are several examples of various substitute commands and addressing combinations:

3s/the/xyz/g

In line 3, change every occurrence of the character trio **the** to **xyz**, even if it occurs in a word such as **weather**.

3s/ the /xyz/g
3s/^the /xyz/
3s/ the$/xyz/

In line 3, change every occurrence of the character trio **the** to **xyz**, provided it is preceded and followed by a space. To test for **the** at beginning and end of line requires two additional commands, one for each case. *ed*, *sed*, and *ex* offer no equivalent to \< and \> that are used in *vi* to delimit a word with whitespace, beginning-of-line, or end-of-line.

⟨*address*⟩**s/the\n/xyz\n/**

Same as above but isolates the word **the** immediately preceding an embedded newline when multiple output lines are present in the current pattern space. Note the use of **\n** in the replacement text to preserve the position of the newline which would otherwise be destroyed.

/the/s/⟨*space*⟩**for**⟨*space*⟩**/xyz/g**

In every line containing the text pattern **the** (even if the pattern is part of a larger word), search for the characters **for** with a space before and after (not at end or beginning of line unless there is a space on both sides of the word). If the pattern exists, replace all five characters with the three characters, **xyz**. If the word **for** is found more than once in the specified sequence, replace every occurrence of the pattern.

/the/s/⟨*space*⟩for⟨*space*⟩/xyz/gw changes

Same as before, but also write the current pattern space to a file named *changes* in the current directory. This technique can be used to create a log file that contains a copy of all changed lines as they appeared after the substitution is complete. For other whole-line-oriented commands, the **w** *file* command form must be used after the command that produced the change. No output occurs on addressed lines if no substitution is made on that line. If no substitutions are made in the entire file that would otherwise be written to file *changes*, the file is created and opened at the beginning of the edit, but when closed at the end, *changes* is a zero-length file.

/the/s/⟨*space*⟩for⟨*space*⟩/xyz/gp

Same as before, but the changed line is printed to standard output if the address matches. As with the **w** flag, no output occurs on addressed lines if no substitution is made on that line.

The **p** flag has no effect unless the **−n** option is used in the original HP-UX *sed* command that started the session because the pattern space is always printed to standard output after it has been edited if **−n** is not specified in the HP-UX command line.

/^125/,/^248/s/[*space tab* **][** *space tab* **]*/**
/g

Starting with the line that begins with **125** as the first three characters in the line through the line that begins with **248**, replace every sequence of one or more contiguous spaces and/or tabs with a single newline (end-of-line). This has the effect of placing all words in the lines (as determined by whitespace separators) in a single column, one word per line.

/^The first time/,$s/lsf.*file3/lsf \/usr\/man* $HOME file3/

Starting with the first line in the file that contains the string **The first time** starting in the first column, and continuing through end of file, whenever a line containing the command **lsf** followed by any number of arbitrary characters in turn followed by the filename **file3** is encountered, replace the entire string (**lsf** through **file3**) with a new text pattern **lsf /usr/man* $HOME file3**. Note the use of \ to protect the / characters in **/usr/man*** from interpretation as delimiters by the editor.

Using Sub-expressions in Substitutions

In the table of regular expressions earlier in this chapter, use of the character pairs \(and \) as delimiters to isolate part or parts of a regular expression as an aid in substitutions was discussed. This obscure and often confusing feature is a very useful mechanism for making certain types of changes in text files such as reversing columns in a table, swapping first and last name in mailing or personnel lists, etc., although its usefulness is not limited to rearranging order of appearance in a line.

Earlier in this chapter, we discussed using sub-expressions in addresses and showed how to identify a line containing one word twice in succession. Since the obvious next task is to fix the unwanted duplication, let us use the same technique to change two words into one.

We determined that two addresses were needed to cover duplicate words at beginning of line and after beginning of line with arbitrary whitespace between words:

/ ^ \([^ *space tab*] . . * \) [*space tab*] [*space tab*] * \ **1** /

and

/ [*space tab*] \([^ *space tab*] . . * \) [*space tab*] [*space tab*] * \ **1** /

This means that two substitute commands must be used; one for each case. For convenience, let us refer to each of the two addresses as *addressA* and *addressB*, respectively (after removing the / character before and after for clarity) in the discussion which follows.

We already know that regular expressions used in addresses are completely independent entities that have nothing in common with the regular expression used to find a pattern match for the substitution, other than they must both match a given line before a substitution can occur in that line.

The safest way to ensure that the matched substitution pattern is the same as the pattern that produced the address match is to use the same expression in both positions in the command line. Thus, using the abbreviated notation specified earlier, the two commands become:

/addressA/**s**/*addressA*/ **\1/**
/addressB/**s**/*addressB*/ **\1/g**

Note the use of the **g** flag after the second command. If two or more repeated word pairs occur in a given line like like this this has has, any successive pairs can be detected and fixed. This eliminates the need to place the commands in a programmed test loop using the **t** and **b** commands described later under flow-of-control commands. Note also the space before the **\1** to preserve spacing between words.

Another situation where using sub-expressions is helpful is in rearranging columns in text such as in a list of last and first names separated by comma and space that must be switched to first and last name separated by a single space. Assuming, for this example, that only the names appear on each line with an optional one or more middle names and no additional non-name text, use a command of the form:

s/\(.*\), \(.*\)/\2 \1/

Of course, this command operates on all lines in the file because no addresses are specified. If the list of names occurs only on a group of lines or on certain isolated lines, proper addressing means must be employed to ensure correct processing.

Transform Command

The transform command (**y**) is used to translate text characters into different characters. It is similar to the substitute command described earlier in that it changes text within the current pattern space, but it does not replace a text pattern with another. Instead, it behaves more like a series of substitute commands that each changes one character globally throughout a line to another without having the ability to "shoot itself in the foot" so to speak by changing a character to another only to have the changed character altered by a subsequent command.

The transform command compares each character in the pattern space with a series of characters contained in the expression called *string_1*, and replaces each match with the character that occupies the same relative position in *string_2*. Thus, the command:

y/abc/cde/

changes every occurrence of **a** in the pattern space to **c**, **b** to **d**, and **c** to **e** for every line in the input file. Like the substitute command, it accepts 0, 1, or 2 addresses. Unlike the substitute command, it does not accept flags at the end of the command. The **g** flag is implied by the very nature of the command. **p** and **w** *filename* must appear as separate commands, if needed.

The transform command can be constructed in any of the following three forms:

- No address performs the transformation on all lines in the input file:

  y */string_1/string_2/*

- A single address performs the transformation on all lines whose line number or text matches *address_1*:

  address_1 **y** */string_1/string_2/*

- A double address performs the transformation on all lines whose addresses fall within the range defined by *address_1* and *address_2*.

  address_1,*address_2* **y** */string_1/string_2/*

A transformation command to convert uppercase to lowercase characters on all lines would resemble the following:

y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/

Of course, this technique is useful for many other situations, including alteration of non-printing ASCII control codes.

Input/Output Commands

The input/output command group is related to reading from and writing to specified external files and printing to standard output. The **write**, **print**, and **list** commands accept 0, 1, or 2 addresses because any number of lines from the input file can be written to a file. The **read** command accepts only 0 or 1 address because the address tells the editor where to place the file when it is read and copied to standard output. The following table lists the commands and their operating characteristics.

Input/Output Commands

Command	No. of Addresses	Description
p print to std out	0, 1, or 2	If the line currently held in the pattern space matches the address or address range for this command, print the pattern space in its present form to standard output, regardless of what changes might be made to the current pattern space by subsequent commands in the command stream.
l list to std out	0, 1, or 2	Same as **p** except that non-printing text characters are printed as their octal equivalent (each octal character code is preceded by a backslash) and long lines are folded into multiple lines not exceeding 72 characters in length (excluding newline). For example, ASCII STX ([CTRL]-[B]) is printed as **\02**, and embedded newlines are printed as a backslash at the end of a line with the remainder of the line (or up to the next newline) printed on the next line.
w ⟨*filename*⟩ Write to file	0, 1, or 2	If the line currently held in the pattern space matches the address or address range for this command, write the pattern space to the file specified by ⟨*filename*⟩ in its present form, regardless of what changes might be made to the current pattern space by subsequent commands in the command stream. If the file is not already open, open the file before writing. Exactly one space character must separate the **w** command from the ⟨*filename*⟩. In any given commands stream, up to 10 filenames can be specified in **w** commands and after **w** flags in **s** editor command lines. If **r** commands are present in the commands stream, this limit must be reduced by one to nine filenames.

(continued next page)

Input/Output Commands (continued)

Command	No. of Addresses	Description
r ⟨*filename*⟩ Read from file	0 or 1	Read the file specified by ⟨*filename*⟩ and copy to standard output after all editing operations have been completed on the current pattern space and it has been printed to standard output. If the pattern space is modified by commands subsequent to the read command, those commands are executed first. The read does not copy *filename* until after the current pattern space is printed to standard out and a new line is to be copied from the input file into the pattern space. Exactly one space character must separate the **r** command from the ⟨*filename*⟩. In any given commands stream, up to nine filenames can be specified in **w** commands and after **w** flags in **s** command lines if one or more **r** commands are also present. If multiple **r** ⟨*filename*⟩ and/or **a** commands are included in a single commands stream, the text produced by the two commands are placed after the current line in the same sequence as the commands that produce the text.

I/O Commands: Operating Characteristics

These commands do not affect the current line in the pattern space. The print and write commands immediately copy the line in the current pattern space to standard output or to the specified file as the case may be, then continue with the commands stream. If the command happens to be the last command in the stream, a new line is read into the pattern space. If it is not the last command, subsequent lines in the stream are executed until the stream has been completed for that line.

On the other hand, the read command does not act immediately. To conserve memory and prevent unnecessary file size limitations, instead of reading the specified file into memory for writing out later, it places the read operation in a tasks queue and waits until processing of the commands stream on the current pattern space line is complete, then copies the specified file to standard output.

If a **w** command is present in the stream, a new file must be opened the first time it is encountered. Since the entire stream is re-executed for each line in the file, the file must remain open until the input is completely processed. In addition, whenever a **r** command is encountered, a file must be opened for the duration of the read. The file is closed as soon as the copy is complete, so it is not necessary to have more than one file open at a time when reading multiple files in a single stream. *sed* allows up to 10 open files at any given time (the HP-UX limit for a single process is 29). This means that if there are no **r** commands in the commands stream, you can specify up to 10 different filenames in conjunction with **w** commands or **w** flags after **s** commands. If you are reading from one or more files with the **r** command, the limit for **w** must be reduced to nine different filenames. Of course, there is no limit to the number of **w** commands and/or flags. Only the number of open filenames is limited.

Using I/O Commands

Here are several examples of how these commands can be used in a typical commands stream:

/Example 1 goes here/r example1
/Example 1 goes here/d

Read file *example1* into output after the line containing the text: "Example 1 goes here". The second command in the stream deletes the original line that matched the address so that it does not appear in the standard output file since it obviously is no longer needed. The order of the commands cannot be reversed because the second command would destroy the line before the address in the first command could match it.

3,20w text_block1

Copy lines 3 through 20 into an external file named *text_block1*. This command can also be grouped with other commands by using the commands-grouping delimiters **{** and **}** to work in conjunction with the **w** *file* flag on substitute commands when creating a log file of changed lines resulting from the edit. It can also be used for other purposes as your needs might dictate.

l

This is a deceptively simple command. It can be used to produce a readable copy of any file containing ASCII control characters. For example, the HP-UX command:

sed -e 'l' source_file >visible_file

produces a new file named *visible_file* that contains the text from *source_file* with control characters in octal value form preceded by a backslash. Thus **\177** represents a DEL character, and a DC2 character (CTRL-R) is listed as **\22**. The horizontal tab character, instead of being listed as **\11**, is displayed as the single character >. Newlines (line-feed) are usually not displayed as an octal sequence. Note that a space is required after the **-e** in the command line.

An alternative to this command is the HP-UX *cat* command using the **-v** option. See *cat*(1) in the *HP-UX Reference* for more information.

Processing Multiple Lines Simultaneously

There are essentially two situations where multiple lines are encountered in the pattern space which is always treated as a single line by *sed*:

- The pattern space is split into multiple lines by a substitute command that includes one or more newlines in the replacement text.
- When processing running text, and part of the text pattern of interest is at the end of one line and the remainder is on the following and possibly subsequent lines.

Both of these conditions require the ability to manipulate part of the pattern space without using the rest. Conducting pattern searches and text manipulation in such situations can become complex and difficult, but is made considerably easier by the availability of three *sed* commands that are used to merge multiple lines into the pattern space and processing only the first part of the pattern space up to the first newline. Note that each command is an uppercase letter representing a very similar standard *sed* command. The commands function as described in the following table.

Multiple-Line Commands for Current Pattern Space

Command	No. of Addresses	Description
N Next line	0, 1, or 2	Read next line from input file and append to current pattern space. A newline character is embedded between the previous pattern space contents and the new line so that pattern matches can extend across the line boundary.
D Delete first part of pattern space	No address	Delete text from beginning of pattern space up to and including the first newline character and restart execution of the commands stream with the first command in the stream. Equivalent to removing the first line in the pattern space. If the newline is also the last character in the pattern space, the next line is read from the input file. Exactly equivalent to **d** if there are no embedded newlines in the pattern space (only newline is at end of pattern space).
P Print first part of pattern space	No address	Print text from beginning of pattern space up to and including the first newline character in its present form to standard output, regardless of what changes might be made to the current line by subsequent commands in the command stream. Does not affect current pattern space contents. Exactly equivalent to **p** if there are no embedded newlines in the pattern space (only newline is at end of pattern space).

These commands are most commonly used in grouped commands using { and } grouping delimiters and in command streams that are controlled by conditional and unconditional branching commands in the flow-of-control group.

Hold and Get Commands

In addition to the pattern space that is used for formal editing operations, *sed* maintains a separate hold space that serves as temporary storage for the current pattern space. You can copy or append the pattern space to the current hold space (copy destroys previous contents), copy or append hold space to current pattern space (again, copy destroys existing pattern space contents), or swap hold and pattern space contents. Hold text in hold area Exchange text with hold area

Command	No. of Addresses	Description
h Copy pattern space to hold area	0, 1, or 2	Copy entire current pattern space contents to hold area. Destroys previous hold area contents.
H Append pattern space to hold area	0, 1, or 2	Append a newline followed by the entire current pattern space contents to the current hold area contents. Preserves existing hold area contents.
g Copy hold area to pattern space	0, 1, or 2	Copy current hold area contents into pattern space. Destroys current pattern space contents.
G Append hold area to pattern space	0, 1, or 2	Append a newline followed by the entire current hold area contents to the current pattern space contents. Preserves existing pattern space contents.
x Exchange pattern space and hold area	0, 1, or 2	Move current pattern space contents to hold area and current hold area contents to pattern space. Swaps the two text blocks without otherwise altering any text in either.

Hold and Get Commands Example

Consider the following text taken from Coleridge:

```
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.
```

Applying the following commands:

```
1h
1s/ did.*//
1x
G
s/\n/  :/
```

produces this result:

```
In Xanadu did Kubla Khan  :In Xanadu
A stately pleasure dome decree:  :In Xanadu
Where Alph, the sacred river ran  :In Xanadu
Through caverns measureless to man  :In Xanadu
Down to a sunless sea.  :In Xanadu
```

Now for an explanation. The first three lines in the command script operate only on line 1 in the source text. The first command copies line 1 from the pattern space into the hold area. The second command substitutes all text starting with the space before the word **did** in the pattern space with nothing, in effect deleting the rest of the line. The third command then exchanges the hold area and pattern space so that the pattern space now contains `In Xanadu did Kubla Khan`, and the hold area contains `In Xanadu`.

The remaining lines affect all lines in the source file. While the original first line is still in the pattern space, the **G** command appends a newline and the hold area contents to the line in the pattern space. The last command is a substitution command that replaces the newline between the original pattern space line and `In Xanadu` with two spaces and a colon. After the last command is applied to the line in the pattern space, it is written to the output. As subsequent lines are read from the input file, the first three commands are skipped since their addresses match only line 1 which has already been processed. The last two commands have no specified address, so they perform their hold-area append and substitution operations on every line.

Flow-of-Control Commands

These commands are used to create program structures when writing *sed* command scripts. They provide conditional branching, unconditional branching, grouping of multiple commands and tying them to a single address or address pair, address inversion (execute command on all lines not addressed by the address on that command), and labels used for branching. Flow control commands Don't (invert address) command Group commands Label command Branching commands

Flow-of-Control Commands

Command	No. of Addresses	Description
! Don't	0, 1, or 2	Reverses the sense of address matching so that the associated command is performed on all lines **except** those that match the line-number or context address(es) appearing on the same line as the !.
{ and } Group commands	0, 1, or 2	Associates two or more commands with a single command-line address field so that multiple related operations can be performed without retesting for address match at each command line. Commands are grouped in a script as follows: ⟨ *address(es)* ⟩ { optional first command first (if not on previous line) or second command ⋮ (additional commands as needed) ⋮ last command } Groups can be nested up to 100 levels deep.
:⟨ *label* ⟩ Define and locate a label	none	Marks a location in the commands stream that can be referred to by **b** (unconditional branch) and **t** (conditional branch) commands. ⟨ *label* ⟩ can be any sequence of seven or fewer characters. If two identical labels are defined in the commands stream, an error is generated and no execution of the commands stream is attempted. Up to 48 labels are allowed in a commands stream. A blank (space or tab) after the colon is optional.

Flow-of-Control Commands (continued)

Command	No. of Addresses	Description
b⟨ *label* ⟩ Unconditional branch	0, 1, or 2	Causes commands stream execution to immediately and unconditionally start execution at the first command following :⟨ *label* ⟩ in the commands stream. If no corresponding ⟨ *label* ⟩ definition can be found at compile time, an error is generated and no execution of the commands stream is attempted. A blank (space or tab) after the colon is optional. If the **b** command has no ⟨ *label* ⟩ argument, the branch is to the end of the commands stream. The current input line in the pattern space is subjected to normal last-command processing (such as being printed on standard output if printing has not been suppressed by a **−n** HP-UX command-line option), a new line is read, and processing resumes with the beginning of the commands stream.
t⟨ *label* ⟩ Conditional branch	0, 1, or 2	Tests to determine whether **any** successful substitutions have been performed on the current pattern space line. If so, it branches to ⟨ *label* ⟩. If not, it continues with the next command in the stream. A substitution flag is set whenever a substitution occurs. It can be cleared by: 1. Reading a new line from the input file, or 2. Executing a **t** command.

Miscellaneous Commands

Two commands are in this group. One prints the number of the addressed line on standard output and the other aborts the editing session. Print line number command Quit command

Command	No. of Addresses	Description
= Line number	1	Print the input-file line number of the line currently in the pattern space. The line is printed as a decimal line number starting in the left column without leading zeros or whitespace on a line by itself. The line number appears on standard output before the line in the current pattern space if it is printed to standard output at the end of the commands stream for the current line.
q Quit	1	Terminate the editing session immediately if the address field on the command line matches the current line in the pattern space. Current pattern space is processed according to normal end-of-commans-stream procedures and the *sed* program terminates (current pattern space is printed on standard output before termination unless suppressed).

Writing Command Scripts 3

This chapter explains how to combine commands that were described in previous topics into useful command scripts for use with the *sed* command. A non-trivial example that looks fairly simple on the surface is used to illustrate the use of *sed*'s capabilities.

Command Script Limitations in Review

There are a few limitations in *sed* scripts that must be carefully followed. Otherwise, you have a very flexible and powerful tool at your fingertips.

- A commands script file can contain up to 100 *sed* commands. Comment lines and script lines that serve as labels are not included in this limit. Multiple, grouped commands that share a common address are treated as separate commands when calculating the 100-command limit.
- When a commands script is implemented as multiple **−e** options in the HP-UX *sed* command line, the maximum number of **−e** options that can be used is determined by the limits stated here for command file scripts plus other factors such as command-line buffer size and other factors, but is generally greater than most users' needs.
- Up to 50 different labels are allowed in a script. Fifty-one labels causes an error.
- Label names can contain up to seven alphanumeric characters. Eight characters produces an error.
- Grouping commands ({ and }) can be nested up to 100 deep.
- Not over 10 files can be open at one time for read/write commands to separate files other than the input file being edited. Only one file is open at any time for read commands, no matter how many may be present in the commands stream.

Arranging Commands in Sequence

The order in which commands are executed can be critically important to obtaining the correct results. For example, grossly incorrect results can occur in either of these two cases:

- A command early in the commands stream alters a pattern in the line currently residing in the pattern space. A later command in the stream which must be able to recognize the pattern in its original unaltered state cannot because the pattern no longer exists due to the earlier change
- A modification of the line by an earlier command in the stream creates a pattern that is recognized by a later command but that should have been ignored.

The number of possible combinations that can lead to incorrect results is virtually endless, so it is impossible to include them all in this tutorial. To simplify the problem as much as possible, you must carefully analyze the commands in your commands stream to ensure that no addressing conflicts and altered text can create an opportunity for *sed* to misinterpret text patterns in the current pattern space during commands execution. This is most easily done by running the commands stream on one or more test files with sufficient rigor to ensure that the stream is working correctly. You may also find it much easier to start with a simple script and build on that until you have all the needed features to accomplish the task at hand.

Commenting Scripts

It can be extremely difficult to interpret an existing command script that someone else wrote or that you once wrote and have forgotten how, especially if you used some exotic "neat tricks" to perform some complicated format conversion (something that *sed* can be very useful for). The task is greatly simplified if you take the time to add comments to your script explaining what the next instruction or group of instructions does.

To add comments, simply start the comment line with a hash mark character (#) and one or more spaces just like when commenting shell scripts. The comment character must be in the first column. Unfortunately, leading whitespace is important in *sed* scripts, so you cannot improve readability by indenting the commands without introducing malfunctions in the script. Here is an example of two script command lines that remove whitespace (space or tab) at the beginning and end of every line in a file:

```
# Remove whitespace at beginning and end of line before continuing.
#
s/[spacetab][spacetab]*$//
s/^[spacetab][spacetab]*//
#
# Comment for next command. . .
#
```

Facts of Life

It is very difficult to represent invisible characters in text so that they are easily identified. HP-UX and similar systems recognize space and tab characters as "blanks", but editors have no single regular expression character that matches both a space and a tab. Thus it becomes necessary to use a space and tab between a pair of square brackets. Since there are no standard typesetting characters that can be appropriately used to represent the two chararacters, and resorting to forms such as ⟨*space*⟩⟨*tab*⟩ are difficult to code and look cluttered when printed, we have used *spacetab* or *space tab* (with a thin space between the two words) to represent a space and a tab together (space first or vice-versa) in a regular expression for text matching purposes.

A Real-World, Non-Trivial Example

sed was used once on a project that required reformatting the *Starbase Reference* manual from inconsistently applied *troff* coding sequences to consistently styled *man* macros. The general layout of the pages was also extensively modified at the same time. *sed* completed about 90% of the work on over 250 pages (over 100 files) in less than four minutes; a task that would have consumed most of a month for a moderately skilled operator using *vi*. This example is loosely based on that experience.

Here is a sample paragraph using conventional *troff*[1] coding:

```
This paragraph contains \fItroff\fR coding where acronyms such as
\s-1HP-UX\s+1 are set in \s-1SMALLER\s+1 typeface and \fBemphasis\fR
as well as \fBcommand options\fR are in \fBboldface\fR.  Most of
these \fItroff\fR coding sequences can be converted into macros
using the \fIman\fR macro package described in the \fIman\fR(5)
entry in the \fI\s-1HP-UX\s+1 Reference\fR.  Much of the work can
be done by a \fIsed\fR script, saving you valuable time.  The
\fIsed\fR(1) manual page entry is mainly for experienced users.
```

When finished, the desired result should look something like this:

```
This paragraph contains
.I troff
coding where acronyms such as
.SM HP-UX
are set in
.SM SMALLER
typeface and
.B emphasis
as well as
.B command options
are in
.BR boldface .
Most of
these
.I troff
coding sequences can be converted into macros
using the
.I man
macro package described in the
```

[1] *troff* is not supported on HP-UX, but equivalent programs are available for HP-UX systems from various third-party software suppliers (for more information, contact your nearest HP Sales and Support Office). *troff* coding sequences and *man* macros are compatible with the *nroff* formatter provided as part of standard HP-UX, although some features cannot be implemented. For example, font changes use highlighting instead, and point size changes cannot be performed by *nroff*. The man macro package is stored in file */usr/lib/macros/an*.

```
.IR man (5)
entry in the
.SM
.I HP-UX
.IR Reference .
Much of the work can
be done by a
.I sed
script, saving you valuable time.  The
.IR sed (1)
manual page entry is mainly for experienced users.
```

This paragraph poses some interesting practical problems. Without launching into a lengthy dissertation, a few explanations are in order so you can understand the logic necessary in forming the correct commands sequence and structure.

troff normally typesets in Roman font (typeface). The **\fI** and **\fB** coding sequences change to *italic* and **bold** fonts, respectively. The **\fR** sequence after the word(s) set in italic or bold is a change back to standard Roman font. *HP-UX Reference* manual entries are referred to by page name followed by section number where the name is in italic and the section number (in parentheses) is in Roman. Acronyms and uppercase character strings such as **HP-UX** tend to look overwhelming when placed in typical text, so *troff* users usually set them in a subdued point size (one point smaller) by using the *man* macro **.SM** instead of preceding the uppercase text with **\s-1** and following it with **\s+1**.

For a first pass attempt, let us collect some basic pieces to start forming the structure.

- First, eliminate all whitespace (space or tab characters) at end of pattern space so that there are no invisible blanks to destroy subsequent editing logic. End-of-line occurs only once, so there is no **g** flag on the command. Use:

```
# Get rid of end-of-line whitespace
s/[spacetab][spacetab]*$//
```

  A similar command is used to remove whitespace at beginning of pattern space (change **$** to ^ and move to front of expression):

```
# Get rid of beginning-of-line whitespace
s/^[spacetab][spacetab]*//
```

- The **\s+1** after acronyms and uppercase words needs to be changed to a newline if it is followed by one or more spaces, or deleted if it occurs at end-of-line. This change the text block in the pattern space which now contains one or more embedded newlines to still be treated as a single line by *sed*, even though the embedded newlines produce a multi-line output when printed. Two commands are required to make the desired changes (two space characters precede the * in the first command):

```
# Change \s+1 followed by 1 or more spaces to newline.
s/\\s+1  */\
/g
# Delete \s+1 at end of pattern space.
s/\\s+1$//
# Delete \s+1 before embedded newline, but keep newline.
s/\\s+1\(\n\)/\1/
```

- Do the same for any **\fR** after a word if it is followed by one or more spaces (convert to newline) or end of line (delete **\fR** at end-of-line). These changes also break the pattern space into additional lines that must be handled carefully in future operations on the pattern space. As before, use two commands (two space characters precede the * in the first command):

```
# Change \fR followed by 1 or more spaces to newline.
s/\\fR  */\
/g
# Delete \fR at end of pattern space.
s/\\fR$//
# Delete \fR before embedded newline, but keep newline.
s/\\fR\(\n\)/\1/
```

- If **\fR** is followed by a comma or period, it must be changed to a space for use with the **.BR** or **.IR** macro. If the comma or period is not at the end of the line, it must be followed by one or more spaces which, in turn must be changed (s) to a newline. If the comma or period is at end-of-line, no newline is needed. As before, two commands for used (note no global flag for end-of-line, unlike within the line and two spaces before * for matching one or more):

```
# Change \fR, and 1 or more spaces to global space comma newline.
s/\\fR,  */ ,\
/g
# Change \fR, at end of line to space comma.
s/\\fR,$/ ,/
# Change \fR. and 1 or more spaces to global space period newline.
s/\\fR.  */ .\
/g
# Change \fR. at end of line to space period.
s/\\fR.$/ ./
```

Time to Test

Now that we have a skeleton, let's try it. Placing the script in file *filesed* and the original paragraph in file *file* then executing:

sed -f filesed file RETURN

we get the following results.

First, the script file in its present form:

```
# Get rid of end-of-line whitespace
s/[spacetab][spacetab]*$//
# Get rid of beginning-of-line whitespace
s/^[spacetab][spacetab]*//
# Change \s+1 followed by 1 or more spaces to newline.
s/\\s+1  */\
/g
# Delete \s+1 at end of pattern space.
s/\\s+1$//
# Delete \s+1 before embedded newline globally, but keep newline.
s/\\s+1\(\n\)/\1/g
# Change \fR followed by 1 or more spaces to newline.
s/\\fR  */\
/g
# Delete \fR at end of pattern space.
s/\\fR$//
# Delete \fR before embedded newline, but keep newline.
s/\\fR\(\n\)/\1/
# Change \fR, and 1 or more spaces to global space comma newline.
s/\\fR,  */ ,\
/g
# Change \fR, at end of line to space comma.
s/\\fR,$/ ,/
# Change \fR. and 1 or more spaces to global space period newline.
s/\\fR.  */ .\
/g
# Change \fR. at end of line to space period.
s/\\fR.$/ ./
```

To clearly show how the pattern space is being handled by the editor, the lines are shown in the form in which they are held in the pattern space followed by the form they would take when sent to standard output. On output, each location in the following lines that contains a **\n** is converted to an end-of-line and the next character following the **\n** appears at the beginning of the following line.

Here is how each line in the file looks while it is still in the pattern space after completion of the last command that affects it:

```
This paragraph contains \fItroff\ncoding where acronyms such as
\s-1HP-UX\nare set in \s-1SMALLER\ntypeface and \fBemphasis
as well as \fBcommand options\nare in \fBboldface .\nMost of
these \fItroff\ncoding sequences can be converted into macros
using the \fIman\nmacro package described in the \fIman\fR(5)
entry in the \fI\s-1HP-UX\nReference .\nMuch of the work can
be done by a \fIsed\nscript, saving you valuable time.  The
\fIsed\fR(1) manual page entry is mainly for experienced users.
```

Here is how the edited file looks when it appears in the output file:

```
This paragraph contains \fItroff
coding where acronyms such as
\s-1HP-UX
are set in \s-1SMALLER
typeface and \fBemphasis
as well as \fBcommand options
are in \fBboldface .
Most of
these \fItroff
coding sequences can be converted into macros
using the \fIman
macro package described in the \fIman\fR(5)
entry in the \fI\s-1HP-UX
Reference .
Much of the work can
be done by a \fIsed
script, saving you valuable time.  The
\fIsed\fR(1) manual page entry is mainly for experienced users.
```

On to the Next Task

So far, so good. But we still have more to do. Now looks like a good time to start attacking the **\fB** and **\fI** patterns. No, we haven't forgotten those **\fR** sequences that are still in there. They are best left in there for now (remember the guy on TV who keeps saying, "Trust me."?).

- Change **\fI** preceded by spaces to a newline followed by a **.I** macro and a space. If the **\fI** is at the beginning of the line already, change it to **.I** and a space. Use two commands [Again, two spaces before the *. Do you remember why?]:

```
# Change \fI after 1 or more spaces to newline, .I and space.
s/  *\\fI/\
.I /g
# Change \fI at beginning of line to .I and space.
s/^\\fI/.I /
```

- Use two commands to do the same for the **\fB** sequences:

```
# Change \fB after 1 or more spaces to newline, .B and space.
s/  *\\fB/\
.B /g
# Change \fB at beginning of line to .B and space.
s/^\\fB/.B /
```

Checking Progress

Adding these commands to the previous script and rerunning it on the original file produces the expected result in the pattern space:

```
This paragraph contains\n.I troff\ncoding where acronyms such as
\s-1HP-UX\nare set in \s-1SMALLER\ntypeface and\n.B emphasis
as well as\n.B command options\nare in\n.B boldface .\nMost of
these\n.I troff\ncoding sequences can be converted into macros
using the\n.I man\nmacro package described in the\n.I man\fR(5)
entry in the\n.I \s-1HP-UX\nReference .\nMuch of the work can
be done by a\n.I sed\nscript, saving you valuable time.  The
.I sed\fR(1) manual page entry is mainly for experienced users.
```

On output, the result looks like this:

```
This paragraph contains
.I troff
coding where acronyms such as
\s-1HP-UX
are set in \s-1SMALLER
typeface and
.B emphasis
as well as
.B command options
are in
.B boldface .
Most of
these
.I troff
coding sequences can be converted into macros
using the
.I man
macro package described in the
.I man\fR(5)
entry in the
.I \s-1HP-UX
Reference .
Much of the work can
be done by a
.I sed
script, saving you valuable time.  The
.I sed\fR(1) manual page entry is mainly for experienced users.
```

Comparing this with the desired text result shown earlier shows that we have made considerable progress toward the desired result, but a few tasks still remain; some easy, some less so.

Cleaning Up What's Left

Changing \s-1 to .SM Macro

First, let's convert all **\s-1** sequences to a **.SM** macro. If it is preceded by a macro such as **.I**, **.B**, etc., it must be converted to a **.SM** by itself on the preceding line. If it is not preceded by a macro, it can be changed to a **.SM** followed by a newline. This is accomplished by using several insert and substitution commands to insert lines and change/delete text within a line as shown in the following script segment that also includes explanatory comments:

```
# Process \s-1 at beginning of pattern space:
s/^\\s-1/.SM \
/
# Process \s-1 preceded by .I, .B, .IR, .BR, etc. at start of pattern space:
#    First, put .SM macro in front of current line on output file.  Assume
#    single space after .I, .B, etc. macro before \s-1.
/^\.[IBR][IBR]* \\s-1/i\
.SM
#    Now, do the same where the line is hidden in the middle of the
#    pattern space.  Instead of using insert, use substitute.
#    Search for \n instead of ^ and use same pattern.  The
#    embedded newline represented by \n gets eaten by the search,
#    and must be replaced by another one in the replacement text.
/\n\.[IBR][IBR]* \\s-1/s/\n\(\.[IBR][IBR]* \)\\s-1/\
.SM\
\1/
#    Now delete \s-1 if line matches same address.
/^\.[IBR][IBR]* \\s-1/s/^\.[IBR][IBR]* \\s-1//
# Now we can process \s-1 in mid-line preceded by one or more spaces or
# tabs.  Use global substitution flag in case of multiple occurrences:
/[space tab][space tab]*\\s-1/s/[space tab][space tab]*\\s-1/\
.SM\
/g
```

Cleaning Up Remaining \fR Sequences

We still have a few **\fR** sequences remaining. They appear to be embedded within basename and section number of *HP-UX Reference* entry titles which occur at end-of-line or within a line. This will require two sets of commands.

To process the end-of-line case (we can handle it first because it is easy and will not sabotage the next command), we must consider end of pattern space and end of line before embedded newline. Two commands are used, one for each case, respectively:

```
# Change \fR in manpage name at end of pattern space to a space.
s/\([a-z][a-z]\)\\fR\(([1-9][A-Z]*)\)$/\1 \2
# Change \fR in manpage name before embedded newline to a space.
s/\([a-z][a-z]\)\\fR\(([1-9][A-Z]*)\n\)/\1 \2
```

The first command searches for any occurrence of **\fR** that is located after two lowercase letters and before a pair of parentheses. The parentheses, in turn, must enclose a number in the range 0-9 followed by any uppercase letter which may or may not be present (***** means zero or more occurrences). The entire pattern as specified must be located at the end of the pattern space (identified by **$** in the search expression).

The second command is very similar, but instead of **$** to indicate end of pattern space, the symbol for an embedded newline (**\n**) is used and placed within the sub-expression delimiters so that it does not have to be added to the replacement string. The replacement string consists of a space followed by the text pattern specified by **\1** which is the text that matched the two parentheses, the number and optional uppercase letter between, and the original newline which is preserved. This method replaces the **\fR** with a space character.

The construction of these two commands specifies that the **\fR** separates what is assumed to be a valid HP-UX command name or manual page entry name based on the presence of two lowercase letters and a valid section number in parentheses (Section 3 entries can have an optional subsection letter as can some commands in Section 1).

The manual entry name in mid-line is handled in much the same way. The only difference is that one or more spaces or tabs must follow the section number and its closing parenthesis and a newline replaces the white space:

```
# Change \fR in manpage name not at newline to space & add newline.
s/\([a-z][a-z]\)\\fR\(([1-9][A-Z]*)\)[space tab][space tab]*/\1 \2\
/g
```

Interim Result

The file, on output, now looks like this:

```
This paragraph contains
.I troff
coding where acronyms such as
.SM
HP-UX
are set in
.SM
SMALLER
typeface and
.B emphasis
as well as
.B command options
are in
.B boldface .
Most of
these
```

```
.I troff
coding sequences can be converted into macros
using the
.I man
macro package described in the
.I man (5)
entry in the
.SM
.I HP-UX
Reference .
Much of the work can
be done by a
.I sed
script, saving you valuable time.  The
.I sed (1)
manual page entry is mainly for experienced users.
```

Now we need to change **.I** and **.B** to **.IR** and **.BR** wherever the **.I** or **.B** is followed by a single word followed by a space and comma or period. This causes the word to be typeset in italic or bold and the punctuation to be set in normal Roman. The space before the comma or period is a delimiter for mixed-font macros and is discarded by the **.IR** or **.BR** macro when the file is formatted prior to printing.

One could easily be tempted to use a command such as these to make the changes:

```
/\.B .* [.,]/s/\.B/&R/
```

and

```
/\.I .* [.,]/s/\.I/&R/
```

These commands search the pattern space for a **.B** or **.I** followed by a space, an arbitrary block of text containing no newlines, another space, and a period or comma. If such a pattern is found, the address matches and the substitute command is executed. The substitute command tells the editor to search for a **.B** (or **.I**) and replace it with the same pattern followed by an uppercase **R**. However, when you run the command, the output shows the lines:

```
    :
as well as
.BR command options
are in
.B boldface .
Most of
    :
```

as the processed replacement for the original third line. We expected the second **.B** to be changed to **.BR**; not the first. If you remember way back in Chapter 2 where we discussed regular expressions, the pattern search occurs through the entire pattern space, and indeed, the address match occurs on the second **.B** which meets the address requirements. However, the substitution begins at the beginning of the pattern space. Thus, the first **.B** in the pattern space is changed to **.BR**, and the second was ignored because no **g** flag was specified after the replacement text argument to the substitute command. We cannot use the **g** flag because the first change would produce "**command**options" instead of "**command options**" like it should. The computer wins again, but the war is not over.

A different approach is obviously in order. Let us pursue some of the methods used earlier in the embedded **\fR** and **\s-1** sequences.

If you observe carefully in the output file as it is currently being processed by the script, the following lines (extracted from the most recent result) are of interest:

```
.B boldface .

.I man (5)

.SM
.I HP-UX
Reference .

.I sed (1)
```

Here is the file as it exists when each line is in the pattern space:

```
This paragraph contains\n.I troff\ncoding where acronyms such as
.SM\nHP-UX\nare set in\n.SM\nSMALLER\ntypeface and\n.B emphasis
as well as\n.B command options\nare in\n.B boldface .\nMost of
these\n.I troff\ncoding sequences can be converted into macros
using the\n.I man\nmacro package described in the\n.I man (5)
entry in the\n.SM\n.I HP-UX\nReference .\n Much of the work can
be done by a\n.I sed\nscript, saving you valuable time.  The
.I sed (1)\nmanual page entry is mainly for experienced users.
```

As you can see, the lines of interest fall at the beginning and elsewhere in the pattern space, so we cannot escape handling all cases. Three situations must be considered:

- Patterns at beginning of pattern space,
- Patterns between embedded newlines, and
- Patterns at end of pattern space.

With some clever handling, we can combine processing of the .I and .B macros. We also know that the space that triggers a need for an **R** suffix on the macro is followed by a period, comma, or left parenthesis. Observation of the resulting file shows that all lines are completely broken down, so there is no need to test for mid-line cases; only embedded newline cases.

Tradeoffs

When you get into a situation like the one currently being discussed where embedded newlines can complicate matters, it is sometimes easier to take the file in its present imperfect form and send it to standard output by not adding any more editing commands to the script. This creates a new file with no embedded newlines within a line, and you can then use a second *sed* command to finish the job. Whether you gain more benefit by taking this "coward's way out" or continuing to process the file in its present form can best be determined by estimating the time it would take to finish the edit using each technique (including allowance for script-writing time) and evaluating the nature of the task. Is it a "one-shot" job, never to be repeated once editing is complete, or is it a task that must be repeated periodically over a longer time period. In the former case, the short-cut, multiple-edit approach may well make more sense. But if you expect to reuse the script periodically, you are likely better off if you clean up the script so that all necessary changes can be handled in a single pass.

To show how the problems we have found so far can be solved in a single script, we will continue as before to finish handling multiple lines in the pattern space. To refresh your memory, here are the lines of interest again:

```
.B boldface .

.I man (5)

.SM
.I HP-UX
Reference .

.I sed (1)
```

The first, second, and last lines look like they should be easy to handle because they occur at the beginning or end of the pattern space or between newlines. They do not occur except where bounded by newlines, or newlines and beginning or end of pattern space. This looks a good opportunity to use sub-expressions. For openers, let's try:

```
# Change .B or .I at beginning of pattern space followed by
# space, word, space and ( , or . to .BR or .IR respectively.
s/^\(\.[IB]\)\( [a-z0-9A-Z][a-z0-9A-Z]* [(,.]\)/\1R\2/
# Do the same for .B or .I after embedded newlines.
s/\(\n\.[IB]\)\( [a-z0-9A-Z][a-z0-9A-Z]* [(,.]\)/\1R\2/g
```

Here is the new result:

```
This paragraph contains
.I troff
coding where acronyms such as
.SM
HP-UX
are set in
.SM
SMALLER
typeface and
.B emphasis
as well as
.B command options
are in
.BR boldface .
Most of
these
.I troff
coding sequences can be converted into macros
using the
.I man
macro package described in the
.IR man (5)
entry in the
.SM
.I HP-UX
Reference .
Much of the work can
be done by a
.I sed
script, saving you valuable time.  The
.IR sed (1)
manual page entry is mainly for experienced users.
```

Comparing this result with the original expected result shown at the beginning of the discussion of this example shows that we have almost accomplished our objective. Two problems remain. The **.SM** macro is not on the same line with the **HP-UX** and **SMALLER** arguments. However, this is not a problem for the man, mm, and ms macros where such structures usually are used. The other case is the second line in *HP-UX Reference*. To work correctly, it needs to be preceded by a **.IR** macro followed by a space.

The first problem is easily solved by a simple substitution of a space for the embedded newline (**\n**) after the **.SM** provided an uppercase character (A-Z) follows the newline. The command to do this is:

```
s/\(\.SM\)\n\([A-Z]\)/\1 \2/g
```

The second problem will probably require a simple search for the actual sequence; that is:

```
s/\(\n\)\(Reference \[.,]\)/.IR &/
```

Putting Scripts in the HP-UX Command Line

When performing complex edits on a large number of files, most users prefer to place editor commands in a separate script so they can be used with the **−f** option to the *sed* command. However, there are also cases, especially in shell scripts, where having the editing commands script embedded within the shell script is a distinct advantage. This is easily accomplished by using multiple **−e** options in a single command but spreading them over several lines by escaping newlines that precede the end of the completed command line.

Note

It is important that you clearly understand that the entire *sed* command must be constructed so that it is correctly treated as a single command line by the HP-UX command interpreter (shell), even though it may span as many as a hundred lines or more in a shell script file. This is accomplished by properly escaping end-of-line on some (but not necessarily all) lines making up the script so that the commands associated with each **−e** option are correctly constructed as single- or multi-line editor commands when they are sent to the *sed* program for compiling after they have been processed by the shell's command interpreter. The techniques for constructing complex command structures is explained in great detail later in this section.

In general, the rules for using multiple **−e** options are the same as for script files. Labels, nesting, and the number of **−e** options per HP-UX command are the same as specified at the beginning of this chapter for command script files except that the available buffer space for command-line processing must not be exceeded. For most applications such limits are not a problem.

Here is an example of how the technique is implemented, based on the first 10 commands in the script that was developed earlier in this chapter:

```
sed -e 's/[space tab] [space tab]*$//'\
 -e 's/^[space tab] [space tab]$//'\
 -e 's/\\s+1  */\
/g'\
 -e 's/\\s+1$//'\
 -e 's/\\fR  */\
/g'\
 -e '# Delete \fR at end of line.
s/\\fR$//'\
 -e 's/\\fR,  */ ,\
/g'\
 -e 's/\\fR,$/ ,/'\
 -e 's/\\fR\.  */ .\
/g'\
 -e 's/\\fR\.$/ ./' file
```

Note how quoting around the argument for each **-e** is used. If the argument requires more than one line, each line is terminated by a backslash (\) when it would also be terminated with a backslash in a separate script file. Likewise, each line where the **-e** argument ends must also be terminated by a backslash after the closing quote for that argument. Note the **-e** argument that also contains a comment; no backslash is used at the end of the comment. No backslash would be used in a command script, so none is used here. The HP-UX shell command interpreter that decodes the command line determines whether the \ escape at end of line applies within the **-e** option and its argument or at the end of it by the presence or absence of the closing single quote (') before the backslash. Anything between the quotes is passed, line-for-line to the *sed* editor for interpretation and compiling. A backslash after the closing quote on any given **-e** option tells HP-UX that the command line continues on the next input line. The HP-UX command interpreter does not terminate interpretation of the command until it encounters a line that does not have a backslash at the end of the line.

Note also how the period in the search pattern in the last two substitute commands is preceded by a backslash. This forces the editor to search for a period following **\fR** instead of any arbitrary character (as it would if using the **.** as a regular expression substitution character).

The name of the file to be edited is then placed after the closing quote on the last **-e** option argument or by itself on a separate line. In the latter case, the preceding line must be terminated by a backslash before the end-of-line just as before. If the filename is placed at the beginning of the last line, it must always be preceded by a space or tab just as when it appears after the argument to the last **-e** option because the escaped

newline (\ followed by end-of-line) is not interpreted as a separator in lieu of a blank by the HP-UX command interpreter.

An example such as the one used here can be included as a small part of a much larger shell script (this technique is used, for example, to produce the Table of Contents and Permuted Index for the *HP-UX Reference*), as a stand-alone script that can be executed as a shell command, or in any combination between the two extremes.

White Space in Scripts

When using multiple **−e** options in a multiple-line format that is being treated as a single *sed* command line, the escaped newline that precedes a given **−e** option is not treated as white space. This means that one or more blanks (space or tab) must be placed either before the backslash (after the closing quote) at the end of the preceding line or before the **−e** appearing at the beginning of the line. Whitespace must also be placed between the end of the last option and the name of the file being edited.

Testing In-Line Scripts

When using scripts in this form, it still saves much time and trouble if you build the script file much as before using an editor such as *vi*, then use the HP-UX command:

```
sh sedfile
```

to test the file for correct operation in its present form. Once a known-good script has been developed, it can then be integrated into a larger script to meet the needs of the larger application.

Including Comments in In-Line Scripts

Comments can be included in in-line scripts that use multiple **−e** options. Simply use a two-line (or more) argument to the **−e** option that resembles the following:

```
sed -e '# Remove end-of-line whitespace
s/[space tab][space tab]*$//'\
 -e '# Remove beginning-of-line whitespace
s/^[space tab][space tab]$//'\
          :
 filename
```

It is not as easy to read as comments in a separate *sed* editor commands script file, but it does provide some indication as to what the command does.

A Large Single-Line Script

Here is a large script based on the example previously developed in this chapter that demonstrates how to construct a non-trivial editing script as a single HP-UX command line:

```
sed -e '# Get rid of end-of-line whitespace
s/[ ][      ]*$//'\
 -e '# Get rid of beginning-of-line whitespace
s/^[        ][      ]*//'\
 -e '# Change \s+1 followed by 1 or more spaces to newline.
s/\\s+1  */\
/g'\
 -e '# Delete \s+1 at end of pattern space.
s/\\s+1$//'\
 -e '# Globally delete \s+1 before embedded newline, but keep newline.
s/\\s+1\(\n\)/\1/g'\
 -e '# Change \fR followed by 1 or more spaces to newline.
s/\\fR  */\
/g'\
 -e '# Delete \fR at end of pattern space.
s/\\fR$//'\
 -e '# Delete \fR before embedded newline, but keep newline.
s/\\fR\(\n\)/\1/'\
 -e '# Change \fR, and 1 or more spaces to global space comma newline.
s/\\fR,  */ ,\
/g'\
 -e '# Change \fR, at end of line to space comma.
s/\\fR,$/ ,/'\
 -e '# Change \fR. and 1 or more spaces to global space period newline.
s/\\fR.  */ .\
/g'\
 -e '# Change \fR. at end of line to space period.
s/\\fR.$/ ./'\
 -e '# Change \fI after 1 or more spaces to newline, .I and space.
s/  *\\fI/\
.I /g'\
 -e '# Change \fI at beginning of line to .I and space.
s/^\\fI/.I /'\
 -e '# Change \fB after 1 or more spaces to newline, .B and space.
s/  *\\fB/\
.B /g'\
 -e '# Change \fB at beginning of line to .B and space.
s/^\\fB/.B /'\
 -e '# Process \s-1 at beginning of pattern space:
s/^\\s-1/.SM \
/'\
 -e '# Process \s-1 preceded by .I, .B, .IR, .BR, etc. at start of pattern
space:
  #    First, put .SM macro in front of current line on output file.  Assume
```

```
#    single space after .I, .B, etc. macro before \s-1.
/^\.[IBR][IBR]* \\s-1/i\
.SM'\
 -e '#    Now, do the same where the line is hidden in the middle of the
#    pattern space.  Instead of using insert, use substitute.
#    Search for \n instead of ^ and use same pattern.  The
#    embedded newline represented by \n gets eaten by the search,
#    and must be replaced by another one in the replacement text.
/\n\.[IBR][IBR]* \\s-1/s/\n\(\.[IBR][IBR]* \)\\s-1/\
.SM\
\1/'\
 -e '#    Now delete \s-1 if line matches same address.
/^\.[IBR][IBR]* \\s-1/s/^\.[IBR][IBR]* \\s-1//'\
 -e '# Now we can process \s-1 in mid-line preceded by one or more spaces or
# tabs.  Use global substitution flag in case of multiple occurrences:
/[   ][      ]*\\s-1/s/[     ][      ]*\\s-1/\
.SM\
/g'\
 -e '# Change \fR in manpage name at end of pattern space to a space.
s/\([a-z][a-z]\)\\fR\(([1-9][A-Z]*)\)$/\1 \2/'\
 -e '# Change \fR in manpage name before embedded newline to a space.
s/\([a-z][a-z]\)\\fR\(([1-9][A-Z]*)\n\)/\1 \2/'\
 -e '# Change \fR in manpage name not at newline to space & add newline.
s/\([a-z][a-z]\)\\fR\(([1-9][A-Z]*)\)[       ][      ]*/\1 \2\
/g'\
 -e '# Change .B or .I at beginning of pattern space followed by
# space, word, space and ( , or . to .BR or .IR respectively.
s/^\(\.[IB]\)\( [a-z0-9A-Z][a-z0-9A-Z]* [(,.]\)/\1R\2/'\
 -e '# Do the same for .B or .I after embedded newlines.
s/\(\n\.[IB]\)\( [a-z0-9A-Z][a-z0-9A-Z]* [(,.]\)/\1R\2/g'\
 file
```

Index

a

c

d

e

f

g

h

i

l

m

n

o

p

r

s

t

Table of Contents

The Ed Editor

Ed is an interactive, line-oriented text editor. Its purpose is to enable you to create ordinary files and to add to, delete, or modify the text in those files.

Creating an Ordinary File

The remainder of this chapter contains several examples illustrating *ed* commands. These examples are more instructive if you try each of them on some text of your own. Thus, create an ordinary file by typing in the commands and text shown below in **bold** (portions of the example text shown below are taken from *A User Guide to the UNIX System*, by Rebecca Thomas and Jean Yates).

```
$ ed testfile
?testfile
a
The ed editor operates in two modes: command mode and
texct entry mode. In command mode, the edytor interprets
your unput as a command. In text entry mode, ed adds
your input to the text located in a special buffer where
ed keeps a copy of the file you are editing. It is \\*.
important to note that ed always makes changes to the
copy of yourrr file in the buffer. The contents of the
original file are not changed until you write the changes
to the file.
.
w
461
q
$
```

Be sure to type in the text exactly as it is shown above. The mistakes are corrected later in the examples.

Getting Acquainted with Ed

Material Covered:

ed	command; invokes *ed* without a file name argument;
ed *file*	command; invokes *ed* with a file name argument;
P	command; enables or disables *ed* prompt (*);
h	command; explains the last question mark given by *ed*;
H	command; enables or disables verbose error messages; explains the last question mark given by *ed*, and all future question marks.

Invoking Ed

Ed can be invoked in one of two ways. The first is to simply type **ed**, followed by [RETURN]. For example,

```
$ ed
```

invokes *ed* without a file name argument. When invoking *ed* this way, you must specify the file you want to edit with a separate command. It is more common to invoke *ed* by typing

```
$ ed filename
```

where *filename* is the name of the file you want to edit. This combines the two separate commands mentioned above into a single command.

Ed responds differently depending on whether or not the file already exists. Try creating a new file called **newfile**:

```
$ ed newfile
?newfile
```

Ed responds with "?newfile", which means that *ed* cannot find that file in your working directory. This is to be expected, since the file does not yet exist. *Ed* is now waiting for your commands to create and edit **newfile**.

If the file already exists, *ed* reads its contents into a buffer named **/tmp/e#**, where # is the number of the process running *ed*. *Ed* then displays a count of the characters contained in that file. You have a file called **testfile** in your working directory. You are probably still in *ed* from the previous example, so type **q**[RETURN] to exit *ed*, then edit **testfile** by typing

```
$ ed testfile
461
```

Ed tells you that **testfile** currently contains 461 characters. Do not exit *ed* this time, but leave it in its current state. The examples that follow pick up where you left off above.

Prompting

One of the most noticeable features of *ed* is its lack of prompts. When you type in a command, *ed* attempts to execute it, and, if successful, *ed* returns silently to you for another command. If an error is encountered, or a command cannot be executed for some reason, *ed* prints a question mark, and then silently waits for you to figure out the problem.

Many people find this silence desirable, but for those who do not, there are commands that make *ed* more friendly. The **P** command causes *ed* to prompt you with an asterisk (*). Executing the **P** command again turns off the prompt. By default, *ed*'s prompt is disabled.

Error Messages

As mentioned above, *ed*'s default error message is a single question mark (?). As you gain experience with *ed*, these question marks become easier to interpret, but for the beginning user, it can be somewhat difficult to discover the problem. Fortunately, *ed* provides commands to eliminate this vagueness. The **h** command explains the last question mark printed by *ed*. The **H** command also explains the last question mark, but also causes a more descriptive explanation of the problem to replace all future question marks. Executing the **H** command again disables the descriptive explanation.

Moving Around in the File

Material Covered:

.	(dot) pointer to the current line;
=	operator; yields line number;
p	command; prints specific lines;
+*n*	operator; increments dot by *n*; default $n = 1$;
-*n*	operator; decrements dot by *n*; default $n = 1$;
$	pointer to the last line of the file;
,	shorthand notation for the range "1,$";
;	shorthand notation for the range ".,$";
k	command; creates a pointer to a specific line;
/ ... /	command; initiates a forward search for the string of characters enclosed between the slashes;
? ... ?	command; initiates a backward search for the string of characters enclosed between the question marks;
.	metacharacter; matches any single character when used in a search string;
n	metacharacter; strips away the special meaning (if any) of the character *n* when used in a search string;
$	metacharacter; when specified as the last character in a search string, matches the string at the end of a line;
^	metacharacter; when specified as the first character in a search string, matches the string at the beginning of a line;
*n**	metacharacter; matches zero or more adjacent occurrences of the character *n* when used in a search string;
[...]	metacharacters; match any one of the characters enclosed between them when used in a search string;
^	metacharacter; stands for "any character except" when specified as the first character inside [...], causing the braces to match any one character *not* enclosed between them;

\{ ... \} metacharacters; match a specified number of occurrences of the single character enclosed between them when used in a search string.

Your position in a file is always relative to a specific line. *Ed* does not provide commands that move you from character to character. There are five commands that enable you to reference specific lines in a file.

Line Pointers

Of the five commands mentioned above, three are pointers to specific lines in the file.

Pointer to the Current Line

Ed maintains a line pointer called dot (.), which always points to the current line in the file. The current line is defined to be the last line affected by an *ed* command. The following table lists some of the more common *ed* operations, and the value of dot after these operations have been performed:

After this operation...	Dot points to...
Invoking *ed*	Last line of file.
Search for pattern	Closest line containing pattern, relative to your previous position.
Delete last line of file	New last line of file.
Delete line(s) other than last line	Line following last deleted.
Appending, inserting, or changing text	Last line entered.
Read from a file	Last line read in.
Write to a file	Your previous position; dot is not changed.
Substitute new text for old	Last line affected by substitution.
Execute a shell command	Your previous position; dot is not changed.
Set a line pointer	Your previous position; dot is not changed.
Any unsuccessful or erroneous command	Your previous position; dot is not changed.

Dot can be used as a line number argument for *ed* commands. Assuming you are still editing **testfile**, type

```
.p
```

to the file.

The **p** command prints specific lines from the *ed* buffer, thus **.p** prints the current line. Note that dot is automatically set to the last line of the file when you first begin editing. You can also specify a range of line numbers with dot. For example,

```
-3,.p
important to note that ed always makes changes to the
copy of yourrr file in the buffer. The contents of the
original file are not changed until you write the changes
to the file.
```

prints the last four lines of the file. Has the value of dot changed? No, because the last line affected by the **p** command was still the last line of the file. Now try

```
-5,-3p
your input to the text located in a special buffer where
ed keeps a copy of the file you are editing. It is \\*.
important to note that ed always makes changes to the
```

which prints the fifth line before dot to the third line before dot. What is dot's value now? Find out by typing

```
.p
important to note that ed always makes changes to the
```

Dot is now set to the last line affected by the previous **p** command.

Note that dot need not be typed when specifying ranges. Whenever *ed* sees the + and - operators, *ed* assumes that they refer to the current value of dot. For example,

```
-2,+2p
your input to the text located in a special buffer where
ed keeps a copy of the file you are editing. It is \\*.
important to note that ed always makes changes to the
copy of yourrr file in the buffer. The contents of the
original file are not changed until you write the changes
```

prints the range of lines from two lines before dot to two lines after dot. Dot is set to the last line printed.

The + and - operators can be used independently to increment or decrement dot by one, respectively. For example, the command

```
--,+p
important to note that ed always makes changes to the
copy of yourrr file in the buffer. The contents of the
original file are not changed until you write the changes
to the file.
```

prints the range of lines from dot decremented by two to dot incremented by one. Also, you can step forward through your text, one line at a time, with a series of plus signs, or step backward with a series of minus signs. Note that [RETURN] is equivalent to +. [RETURN] increments dot by one and prints the resulting current line.

The **p** command provides one other shortcut. Whenever a line number, or one or more operators pointing to a line, appear on a line by themselves, the **p** command is assumed. Some examples are:

```
8
original file are not changed until you write the changes
---
ed keeps a copy of the file you are editing. It is \\*.
++
copy of yourrr file in the buffer. The contents of the
```

If a range appears on a line by itself, only the last line of the range is printed. For example,

```
-,+
original file are not changed until you write the changes
```

You can find out the current value of dot by typing

```
.=
8
```

which tells you that dot is currently pointing to the eighth line of the file.

Note that you cannot manually set the value of dot. A command like

```
.=6
?
```

produces an error. *Ed* reserves to itself the right to change the value of dot, although you may indirectly change dot's value through *ed* commands.

Pointer to the Last Line

Ed also maintains a pointer, called **$**, which always points to the last line of the file. For example,

```
$
to the file.
```

prints the last line of the file. **$** can also be used in ranges, as in

```
1,$-6p
The ed editor operates in two modes: command mode and
texct entry mode. In command mode, the edytor interprets
your unput as a command. In text entry mode, ed adds
```

which prints the first three lines of **testfile**. Also,

```
+4,$p
copy of yourrr file in the buffer. The contents of the
original file are not changed until you write the changes
to the file.
```

prints the last three lines of the file. Note that the + and - operators can apply to **$** only when **$** is explicitly typed. By themselves, + and - always apply to dot.

You can find out the value of **$** by typing

```
$=
9
```

which tells you that the ninth line is the last line in the file. Note that = does not change the value of dot.

The value of **$** changes only when a command creates a new last line. **$** is not user-settable.

Because the "1,$" and ".,$" ranges are so commonly used when editing with *ed*, *ed* provides a shorthand notation for each range. The comma can be used in place of "1,$", so that **,p** prints all the lines in the file. Also, the semicolon means the same thing as ".,$", so **;p** prints all the lines from the current line to the end of the file.

Setting Pointers to Lines

The **k** command creates a pointer to a specific line, so you can reference that line without knowing its line number. The pointer name must be a lower-case letter. Creating a pointer does not change the value of dot. For example,

```
.
to the file.
-4ka
-2kb
.
to the file.
```

creates two pointers, a and b, which point to the fourth line before dot, and the second line before dot, respectively. Note that dot does not change.

To reference a line with a line pointer you have created, precede its letter name with a single quote ('), as in

```
'a,'bp
ed keeps a copy of the file you are editing. It is \\*.
important to note that ed always makes changes to the
copy of yourrr file in the buffer. The contents of the
```

which prints all lines from the line pointed to by a to the line pointed to by b.

A pointer set by the **k** command always points to the same line, even if that line's line number changes. Thus, the **k** command does not create pointers to specific line numbers, but to specific lines.

Once a pointer has been created, the only way to delete it is to delete the line it points to. Otherwise, that pointer continues to exist until your editing session is over. You can, however, re-assign a pointer to another line, as in

'ap
ed keeps a copy of the file you are editing. It is *.
2ka
'ap
texct entry mode. In command mode, the edytor interprets

which re-assigns a to the second line of the file.

You can find out the current line number of a pointer by typing

'a=
2
'b=
7

which tells you that a is currently pointing to line number 2, and b is currently pointing to line number 7.

Searching for Strings

Ed provides a facility which enables you to search for a particular string of characters in your file. A string of characters searched for in this manner is called a *pattern.*

Forward Searches

To initiate a forward search, enclose the pattern between two slashes, and press [RETURN]. For example,

/unput/
your unput as a command. In text entry mode, ed adds

searches for the pattern "unput". If the pattern is found, dot is set to the line containing the pattern, and the line is printed on your screen. An unsuccessful search looks like this:

/bob/
?

The value of dot is unchanged.

Ed searches forward in your file, starting with the line following the current line. If your pattern has not been found by the time *ed* gets to the end of the file, *ed* wraps around to the beginning of your file and continues looking. *Ed* searches until the pattern is found, or until *ed* reaches the line prior to the starting line of the search.

Backward Searches

You can search backwards in your file by enclosing the pattern between two question marks. For example,

```
?file?
to the file.
```

searches backwards from the current line, looking for a line containing the string "file". *Ed* found the pattern after wrapping around to the end of the file.

Repeating a Search

Ed remembers the last pattern that was matched. Thus, if you want to repeat a search, you simply type two slashes or question marks. The pattern itself need not be re-typed. For example,

```
?file?
original file are not changed until you write the changes
??
copy of yourrr file in the buffer. The contents of the
??
ed keeps a copy of the file you are editing. It is \\*.
```

initiates a backward search for the pattern "file", then finds the next two instances of "file". Note that a repeated search need not be in the same direction as the initial search. For example,

```
/buffer/
copy of yourrr file in the buffer. The contents of the
??
your input to the text located in a special buffer where
```

initiates a forward search for "buffer", then repeats the search backwards.

Line Number Arithmetic with Searches

The + and - operators can be used with searches to position yourself at specific lines. For example,

```
/note/+
copy of yourrr file in the buffer. The contents of the
```

searches forward for a line containing "note", and positions you on the following line. Also,

```
?text?
your input to the text located in a special buffer where
??--
The ed editor operates in two modes: command mode and
```

searches backwards for the second line containing "text", and positions you two lines before it.

Note that, although searches have wrap-around capabilities, the + and - operators do not. Thus, an error results if a + or - operator attempts to increment or decrement dot to values greater than **$**, or less than one.

The = operator can be used with forward and backward searches to find the line number referred to by the search, as in

```
/unput/=
3
```

Note that dot is not set to the line containing "unput" in the last example, because = does not change the value of dot.

Using Metacharacters With Searches

There are several characters that have special meaning within the context of a search. These characters, consisting of ., *, [,], ^, $, \, \{, and \}, are called metacharacters.

The . metacharacter matches any single character except a new-line. Thus, the search

```
/.nput/
your unput as a command. In text entry mode, ed adds
//
your input to the text located in a special buffer where
```

first matches "unput" in line 3, and then, when repeated, matches "input" in line 4.

The * metacharacter matches zero or more occurrences of the character immediately preceding it. For example,

/your*/
ed keeps a copy of the file you are editing. It is *.

matches "you" in the line displayed. *Ed* stops searching when it finds the first string of characters that matches the given pattern. Thus, "your" or "yourrr" can also be matched with the above search, depending on the current line when the search is initiated.

The last example shows that, even though an "r" is explicitly typed in **/your*/**, there need not be an "r" in the string of characters that are actually matched. This is because zero occurrences of the preceding character is considered a legal match when the asterisk is used. Keeping this in mind, consider the search **/r*/**. Is it useful? No, because zero or more r's can be found on every line in the file. If you want to search for one or more r's, type **/rr*/**.

The \{ and \} metacharacters enable you to control how many occurrences of a particular character are matched. For example, the search **/g\{4\}/** finds a string of four g's. The integer between the two metacharacters specifies how many instances of the preceding character are to be matched. Note that this construct matches *exactly* four g's, not four or more. Thus, "yourrr" can be matched by

/r\{3\}/
copy of yourrr file in the buffer. The contents of the

If you put a comma after the integer, the \{ ... \} construct matches *at least* the specified number of occurrences. For example, **/33.3\{4,\}/** matches "33.", followed by at least four 3's. Finally, two integers separated by a comma can be placed in the \{ ... \} construct to define an inclusive range which specifies the number of occurrences to match. An example is **/-13\{2,5\}1-/**, which matches -1331-, -13331-, -133331-, or -1333331-.

The [and] metacharacters match any one of the characters enclosed between them. For example, **/h[iau]t/** matches "hit", "hat", or "hut". A range of characters can be specified by typing the beginning and ending character of the range, separated by a minus sign. An example is **/[a-zA-Z][0-9][0-9]*/**, which searches for a single upper- or lower-case letter, followed by one or more digits (the * applies only to the [...] construct immediately preceding it). The minus sign loses its special meaning within the [...] construct if it occurs at the beginning (after an initial ^, if any), or at the end of the character list.

If the first character after the left bracket is a circumflex (^), then the [...] construct matches any single character *not* included between the brackets. For example, **/[^0-9][^0-9]*/** matches one or more occurrences of any character except a digit. The ^ has special meaning in the [...] construct only when it is the first character after the left bracket.

Note that the metacharacters **.**, **[**, ****, **$**, **\{**, **and** **\}** have no special meaning when listed within the [...] construct. Also, the right bracket does not terminate the construct if it is the first character listed after the left bracket (after an initial ^, if any). For example, **/[]a-r]/** searches for a single right bracket, or a lower-case letter in the range a through r.

The ^ is also special when typed at the beginning of a string within a search, and requires that the string be matched at the beginning of a line. For example,

/^ed/
ed keeps a copy of the file you are editing. It is *.

searches for a line beginning with "ed". The ^ is special only when typed at the beginning of a search string. If ^ is embedded in a pattern, or if it is the only character in the pattern, it is matched literally.

The various ways to use ^ can be illustrated with **/^^[^a-z]/**. The first ^ means "match the following pattern at the beginning of a line". The second ^ is literal; it has no special meaning. The third ^, as the first character inside the brackets, means "match any single character except". Thus, this search looks for a ^, followed by any single character except a lower-case letter, occurring at the beginning of a line.

The **$** metacharacter is special when typed at the end of a string within a search, and requires that the string be matched at the end of a line. For example,

/and$/
The ed editor operates in two modes: command mode and

searches for a line ending with "and". Also, **/^TEST$/** searches for a line consisting of the single word "TEST".

The **$** is special only when typed at the end of a search string. When embedded in the string, the **$** is matched literally.

The \ (backslash) metacharacter is used to strip away the special meaning associated with a metacharacter. This is useful when you need to match a metacharacter literally in a string. To strip away the special meaning of a metacharacter, simply precede it with \. For example,

/*\.$/
ed keeps a copy of the file you are editing. It is *.

matches the string "*." at the end of a line. Note that \ itself must also be preceded with \ to be matched literally. If you attempt to match the string without using the \ (as in **/*.$/**), *ed* interprets the search to mean "search for zero or more occurrences of a backslash followed by any single character at the end of a line", which is obviously not what you want. Also,

/file\.$/
to the file.

matches "file." at the end of a line. If you are ever in doubt about whether or not a character has special meaning, it is safe to precede it with \ just to be sure. If the character has no special meaning, then the \ is ignored.

Adding, Deleting, and Correcting Text

Material Covered:

l	command; list specific lines;
n	command; print lines with line numbers;
a	command; append lines of text after current line;
i	command; insert lines of text before current line;
d	command; delete lines of text;
c	command; change lines of text;
m	command; move lines of text;
t	command; copy lines of text;
j	command; join lines together;
s	command; substitute new text for old text;
g	command; global; perform command list on selected lines of entire file;
G	command; interactive global; on each line selected in the entire file, perform a user-specified command;
v	command; global; perform command list on all lines *not* selected in the entire file;
V	command; interactive global; on each line *not* selected in the entire file, perform one user-specified command;
u	command; reverse the most recent modification to the buffer;
\(...\)	metacharacters; used in left-hand side of **s** command to break up pattern into pieces that can be referenced individually;
%	metacharacter; used in right-hand side of **s** command to duplicate right-hand side of most recent **s** command;
&	metacharacter; used in right-hand side of **s** command to duplicate left-hand side of same **s** command.

Printing Lines

Besides **p**, there are two other commands that enable you to print specific lines in the *ed* buffer. The **l** (list) command is similar to **p**, but gives you slightly more information. The **l** command enables you to see characters that are normally invisible. Backspace and tab are represented by overstrikes, and other invisible characters, such as bell, vertical tab, and formfeed, are represented by ***\nnn***, where *nnn* is the octal equivalent of the character in the ASCII character set.

The **l** command also breaks long lines into smaller lines of 72 characters each. Thus, if you have lines of text in a file that are longer than 72 characters, **l** breaks them down into 72-character lines so they can fit on your screen. A \ is printed at the end of each line that is broken.

Print out the contents of **testfile** with the **l** command, and look for any invisible characters:

```
,l
The ed editor operates in two modes: command mode and
texct entry mode. In command mode, the edytor interprets
your unput as a command. In text entry mode, ed adds
your input to the text located in a special buffer where
ed keeps a copy of the file you are editing. It is \\*.
important to note that ed always makes changes to the
copy of yourrr file in the buffer. The contents of the
original file are not changed until you write the changes
to the file.
```

If you did not make any typing errors that could produce invisible characters, the output looks as shown above. Note that a carriage return and a line feed are not considered invisible, since the placement of text on your screen indicates their presence.

Since some invisible characters can cause strange terminal behavior, you almost always want to eliminate them from your text. This is where the **l** command can save you time and effort by making these characters visible.

The **n** (number) command also enables you to print specific lines, but differs from **p** and **l** in that each line is preceded by its line number and a tab character. Try printing out the contents of **testfile** with **n**:

,n
1 The ed editor operates in two modes: command mode and
2 texct entry mode. In command mode, the edytor interprets
3 your unput as a command. In text entry mode, ed adds
4 your input to the text located in a special buffer where
5 ed keeps a copy of the file you are editing. It is *.
6 important to note that ed always makes changes to the
7 copy of yourrr file in the buffer. The contents of the
8 original file are not changed until you write the changes
9 to the file.

Note that the line numbers and tab characters are display enhancements only, and do not become part of the text in the *ed* buffer.

The **p** command is the most common command used to print lines in the *ed* buffer. Keep in mind, however, that wherever it is legal to use the **p** command, the **l** and **n** commands may also be used. The **l** and **n** commands leave dot pointing to the last line printed.

Appending Text

The **a** (append) command appends one or more lines of text after the specified line. By default, the lines of text are added after line dot. Dot is left pointing to the last line appended. After the **a** command is typed, everything you enter is appended to the specified line. To stop appending text, type a period at the beginning of a line, all by itself. This terminates the **a** command, and returns you to command mode. For example,

```
0a
The ed editor is a simple, easy-to-use text editor.
.
1,3p
The ed editor is a simple, easy-to-use text editor.
The ed editor operates in two modes: command mode and
texct entry mode. In command mode, the edytor interprets
```

The **a** command is one of the few *ed* commands that accepts 0 as a line number, enabling you to add text to the beginning of the file, as above. Note that the period at the beginning of an empty line terminates the appended text. The following example can easily occur by forgetting to type the terminating period (*do not try this example!*):

```
$a
It is always comforting to know that your original
file remains intact until you are sure you want to
change it.
1,$p
$-4,$p
;l
.
$-7,$p
original file are not changed until you write the changes
to the file.
It is always comforting to know that your original
file remains intact until you are sure you want to
change it.
1,$p
$-4,$p
;l
```

This poor user typed in the three lines of text that he wanted to append to the end of his file, and then attempted to print out the results. *Ed*, however, was still appending text, and calmly added the user's commands to the file. The user finally realized his mistake, typed the solitary period, and printed out the last eight lines of his file, three of which were the three commands he attempted to execute. The moral of the story is: *REMEMBER THE PERIOD!*

If you type the **a** command and then change your mind, simply type a solitary period on the next line. This terminates the **a** command and adds no lines to the file. Dot is left pointing to the line you specified when you typed the **a** command.

Inserting Text

The **i** (insert) command is similar to the **a** command, except that the added text is inserted before the specified line. By default, the added text is inserted before line dot. Dot is left pointing to the last line inserted. Like the **a** command, the inserted text is terminated by a solitary period at the beginning of a line. For example,

```
2i
Also, it takes very little time to learn.
.
1,3p
The ed editor is a simple, easy-to-use text editor.
Also, it takes very little time to learn.
The ed editor operates in two modes: command mode and
```

If you type the **i** command and then change your mind, simply type a solitary period on the next line. This terminates the **i** command and adds no lines to the file. Dot is left pointing to the line you specified when you typed the **i** command.

Deleting Text

The **d** (delete) command deletes one or more lines of text from the file. If no lines are specified, line dot is deleted. After a deletion, dot is left pointing to the line following the last line deleted. If the last line of the file is deleted, dot points to the new last line. For example,

```
$d
a
on top of the original contents of your file.
.
$-1,$p
original file are not changed until you write the changes
on top of the original contents of your file.
```

The current last line is deleted, and a new one is typed in its place using the **a** command. The **a** command is used because dot is left pointing at the new last line after the deletion. Thus, it is convenient to append after dot to create the desired last line.

The **d** command can delete several lines at once by specifying a range of lines, as follows:

```
3,6d
,p
The ed editor is a simple, easy-to-use text editor.
Also, it takes very little time to learn.
ed keeps a copy of the file you are editing. It is \\*.
important to note that ed always makes changes to the
copy of yourrr file in the buffer. The contents of the
original file are not changed until you write the changes
on top of the original contents of your file.
```

This shows that **testfile** currently contains 7 lines of text, since lines 3 through 6 have been deleted.

Undoing Commands

The **u** (undo) command reverses the effect of the most recent command that made a change to any of the text in the buffer. Use it now to restore the four lines you just deleted:

```
u
,p
The ed editor is a simple, easy-to-use text editor.
Also, it takes very little time to learn.
The ed editor operates in two modes: command mode and
texct entry mode. In command mode, the edytor interprets
your unput as a command. In text entry mode, ed adds
your input to the text located in a special buffer where
ed keeps a copy of the file you are editing. It is \\*.
important to note that ed always makes changes to the
copy of yourrr file in the buffer. The contents of the
original file are not changed until you write the changes
on top of the original contents of your file.
```

Note that the **u** command reverses only the most recent command that modified text. Commands that have been succeeded with one or more other commands cannot be reversed with **u**. Besides **d**, **u** also reverses the **a**, **i**, **c**, **g**, **G**, **v**, **V**, **j**, **m**, **r**, **s**, and **t** commands. Dot is left pointing to the last line affected by the reversal.

Changing Lines

The **c** (change) command replaces one or more lines with the text you specify. The **c** command is a combination of the **d** and **i** commands, in that the specified lines are deleted, and the text you type in is inserted in their place. Like the **a** and **i** commands, the replacement text is terminated with a solitary period at the beginning of a line. Dot is left pointing to the last line of replacement text typed in. For example,

```
1,2c
The ed editor is easy to learn and easy to use.
.
1,3p
The ed editor is easy to learn and easy to use.
The ed editor operates in two modes: command mode and
texct entry mode. In command mode, the edytor interprets
```

In this example, the first two lines are deleted and replaced with a single line. Of course, you can also replace a single line with several lines, as in

```
2c
It was designed to enable the user to get his work done
with the least possible amount of interference from the
editor. This is evident in the lack of prompts and the
curt error messages.
The ed editor operates in two modes: command mode and
.
1,/texct/p
The ed editor is easy to learn and easy to use.
It was designed to enable the user to get his work done
with the least possible amount of interference from the
editor. This is evident in the lack of prompts and the
curt error messages.
The ed editor operates in two modes: command mode and
texct entry mode. In command mode, the edytor interprets
```

which replaces the second line of the file with five lines.

If you type the **c** command and then change your mind, simply type a solitary period at the beginning of the next line. This terminates the **c** command with no changes made, and leaves dot pointing to the first line you specified when you typed the **c** command.

Moving Lines

The **m** (move) command moves one or more lines to a new position in the file. By default, **m** moves line dot. Dot is left pointing to the last line moved. For example,

```
2,5m$
,p
The ed editor is easy to learn and easy to use.
The ed editor operates in two modes: command mode and
texct entry mode. In command mode, the edytor interprets
your unput as a command. In text entry mode, ed adds
your input to the text located in a special buffer where
ed keeps a copy of the file you are editing. It is \\*.
important to note that ed always makes changes to the
copy of yourrr file in the buffer. The contents of the
original file are not changed until you write the changes
on top of the original contents of your file.
It was designed to enable the user to get his work done
with the least possible amount of interference from the
editor. This is evident in the lack of prompts and the
curt error messages.
```

which moves lines two through five to the end of the file. Note that **m** appends the moved lines after the specified line. Thus, line number zero is legal as a destination line number, enabling you to move lines to the beginning of the file. The destination line cannot be one of the lines being moved.

Note that the **m** command, as well as any command that accepts line number arguments, accepts pattern searches and line pointers (set by the **k** command) to reference specific lines. For example, **2,/user/++m$** has the same effect as **2,5m$** in the previous example. Using pattern searches and line pointers becomes more valuable when you edit large files.

Copying Lines

The **t** command copies one or more lines and places the copy at a specified location in the file. By default, **t** copies line dot. Dot is left pointing to the last line copied, in its new location. For example,

```
1t$
.-4,$-1t1
,p
The ed editor is easy to learn and easy to use.
It was designed to enable the user to get his work done
with the least possible amount of interference from the
editor. This is evident in the lack of prompts and the
curt error messages.
The ed editor operates in two modes: command mode and
texct entry mode. In command mode, the edytor interprets
your unput as a command. In text entry mode, ed adds
your input to the text located in a special buffer where
ed keeps a copy of the file you are editing. It is \\*.
important to note that ed always makes changes to the
copy of yourrr file in the buffer. The contents of the
original file are not changed until you write the changes
on top of the original contents of your file.
It was designed to enable the user to get his work done
with the least possible amount of interference from the
editor. This is evident in the lack of prompts and the
curt error messages.
The ed editor is easy to learn and easy to use.
```

This example copied the first line and moved it to the end of the file. Then, the four lines before the new last line were copied and moved after the first line of the file, producing the text shown above.

The only difference between the **m** and **t** commands is that **t** copies the indicated lines and moves them to a new position, leaving the original lines intact. The **m** command moves the specified lines from their original position to a new position. No new text is created.

Modifying Text Within a Line

The **s** (substitute) command is the only *ed* command that enables you to change one or more characters within a line, without having to type the line over again. By default, **s** modifies text on line dot. Dot is left pointing to the last line in which a modification has occurred.

The **s** command enables you to correct the mistakes in your file. Of course, you could use the **d** and **i** commands and re-type each line containing an error, but that is more work than is necessary. For example,

/texct/
texct entry mode. In command mode, the edytor interprets
s/texct/text/p
text entry mode. In command mode, the edytor interprets

All **s** command lines are of the form

s/*replace this/with this/*

Thus, the above example first searches for the line containing the string "texct", and then replaces "texct" with "text" on that line. Note that the **p** command is appended to the **s** command to verify that the intended substitution took place.

Note that the pattern search in the previous example can be included on the **s** command line. The **s** command accepts one line number, to perform a specific replacement on a single line, or two line numbers separated by a comma, to perform a replacement on a range of lines. For example,

/unput/s//input/p
your input as a command. In text entry mode, ed adds

which searches for the pattern "unput" and replaces it with "input". Another feature is illustrated in the above example. Note that the *replace this* portion of the **s** command is empty. This is because the *replace this* portion of the **s** command is a pattern search, just like those discussed under *Searching for Patterns.* You recall from that discussion that *ed* remembers the last pattern you searched for. Thus, since "unput" is the last pattern you searched for, it need not be re-typed in the **s** command. *Ed* remembers the pattern and supplies it for you.

Metacharacters can be used in the **s** command. The *replace this* portion recognizes all the metacharacters discussed under *Searching for Patterns*, plus two additional metacharacters, \ (and \). These two metacharacters are used to break up the *replace this* portion into pieces that can be referenced individually. For example, in line 1 of the file, suppose you want to interchange the phrases "easy to learn" and "easy to use". The obvious way to do that is to retype the entire line, but there is an easier way:

```
1s/\ea.*rn\) and \(ea.*se\)/\2 and \1/p
The ed editor is easy to use and easy to learn.
```

Although it is hard to read, it is handy to be able to define pieces of patterns and rearrange them in the *with this* portion. In the above example, the entire *replace this* portion matches "easy to learn and easy to use". The first \(... \) matches "easy to learn", and the second \(... \) matches "easy to use". These pieces are referred to in the *with this* portion with *n*, where *n* refers to the n-th occurrence of a \(... \) pair in the *replace this* portion, counting from the left. Thus, the *with this* portion interchanges the two pieces defined in the *replace this* portion.

Here is another example. Suppose you have a file containing information like

```
Alderson, Mike
Anderson, David
Belford, John
Donally, Kyle
    .
    .
    .
```

and you want to rearrange each name so that the first name is first, followed by the last name. Retyping each line could take forever, but the task is easy using the \(and \) metacharacters. The command

```
,s/\([^,]*\), \(.*\)/\2 \1/
```

does the job. The first \(... \) pair matches any number of characters except a comma – the last name. The comma-space between each last and first name is explicitly matched. Finally, the second \(... \) pair matches any number of any characters – the first name. These pieces are rearranged in the *with this* portion.

Note that the two portions of an **s** command do not have to be delimited by slashes. You can use any character except a space or a new-line, as long as you use the same character throughout the command line. For example, the previous example can be made a bit more clear by using a capital o as the delimiter:

```
,sO\([^,]*\), \(.*\)O\2 \1O
```

You must be careful to choose a delimiter that is not already used in the **s** command line.

The *with this* portion of the **s** command recognizes only the \ metacharacter, plus two new metacharacters, **&** and **%**. All other metacharacters previously discussed are interpreted literally in this portion.

The **&** metacharacter is recognized only in the *with this* portion, and stands for whatever is matched by the pattern in the *replace this* portion. For example,

```
2s/done/& quickly/p
It was designed to enable the user to get his work done quickly
```

The **&** stands for whatever pattern is matched in the *replace this* portion, so it stands for "done" in this example. Thus, this example replaces "done" with "done quickly". As another example, first add the line "ed is great" to the end of the file:

```
$a
ed is great
.
```

Now use **&** to create two sentences out of one:

```
$s/.*/&? &!/p
ed is great? ed is great!
```

The **&** must be preceded by *e to be interpreted literally.

The % is also recognized only in the *with this* portion, and stands for whatever was specified in the *with this* portion of the last **s** command that was executed. For example,

1s/ed editor/ed text editor/p
The ed text editor is easy to use and easy to learn.
/ed editor/s//%/p
The ed text editor operates in two modes: command mode and
//s//%/p
The ed text editor is easy to learn and easy to use.

In the first **s** command, the *with this* portion has to be explicitly typed out. Thereafter, a % is the only character appearing in the *with this* portion, and stands for "ed text editor". Since the replacement text is the same for the remaining **s** commands, it does not need to be re-typed. Note also how **ed**'s pattern memory is utilized, especially in the last **s** command above.

The % is special only when it is the only character in the *with this* portion. If % is included in a string of one or more characters, it is no longer special. You can also precede the % with a \ to cause literal interpretation.

Now that you know all about the **s** command, you can go through and fix the remaining errors in your file. Here are some suggestions:

/edy/s//edi/p
text entry mode. In command mode, the editor interprets
+3s/*\.//p
ed keeps a copy of the file you are editing. It is
/yourrr/s//your/p
copy of your file in the buffer. The contents of the

Note that, in the second **s** command above, the *with this* portion is empty. This is legal, and is often used when you want to replace erroneous text with nothing at all.

Finally, note that the **s** command operates only on the first occurrence of a pattern on a specified line. Thus, if there are two or more patterns on a line that are identical to the pattern specified in the *replace this* portion, only the first occurrence is actually replaced. The **s** command must be re-executed once for each additional pattern that is to be replaced on the same line.

The **s** command must replace text on at least one of the addressed lines, or *ed* prints a question mark.

Making Commands Effective Globally

The **g** (global) command is used to execute one or more commands on several lines. The lines on which the commands are to be executed are usually specified by pattern searches. The form of a **g** command is

x,*y***g**/*pattern*/*command list*

where *x* and *y* are optional line number arguments, *pattern* is the pattern to be searched for, and *command list* is the list of one or more commands to be executed on each line containing *pattern*. If *x* and *y* are missing, "1,$" is assumed.

The **g** command first marks every line containing the specified pattern. Then, dot is successively set to each marked line, and the list of commands is executed. If only one command is specified, it is placed on the same line as the **g** command. If several commands are specified, the first command is placed on the same line as the **g** command, and all other commands are placed on the following lines. Every line of a multi-line command list is terminated by \ except the last. Ending a line with \ in this way quotes the following new-line, and hides it from the **g** command, thus preventing the new-line from terminating the **g** command prematurely. If no commands are specified, the **p** command is assumed. Any command except **g**, **G**, **v**, and **V** can be used in the command list.

The **g** command can be used as a modifier for the **s** command, enabling the **s** command to replace all the occurrences of a particular pattern on a line, instead of just the first. For example,

```
$s/ed/The & editor/gp
The ed editor is great? The ed editor is great!
```

which replaces both instances of "ed" on the last line with "The ed editor". The **g** command is often used with the **s** command in this way to avoid having to repeat the **s** command once for every additional pattern you want to change on a line. Note that, if the **p** command is omitted, the line is not printed after the substitution is done.

The **g** command becomes more powerful when you specify more than one command to be executed. For example, suppose that you want to change every instance of the string "ed" to "ED", and then mark every line on which the substitution occurs by preceding the line with a series of asterisks. This can be done by typing

```
g/ed/s//ED/g\
i\
***

,p
***
The ED text EDitor is easy to use and easy to learn.
***
It was designED to enable the user to get his work done quickly
with the least possible amount of interference from the
***
EDitor. This is evident in the lack of prompts and the
curt error messages.
***
The ED text EDitor operates in two modes: command mode and
***
text entry mode. In command mode, the EDitor interprets
***
your input as a command. In text entry mode, ED adds
***
your input to the text locatED in a special buffer where
***
ED keeps a copy of the file you are EDiting. It is
***
important to note that ED always makes changes to the
copy of your file in the buffer. The contents of the
***
original file are not changED until you write the changes
on top of the original contents of your file.
***
It was designED to enable the user to get his work done
with the least possible amount of interference from the
***
EDitor. This is evident in the lack of prompts and the
curt error messages.
***
The ED text EDitor is easy to learn and easy to use.
***
The ED EDitor is great? The ED EDitor is great!
```

This example, though not very useful, illustrates how the **g** command can be used to perform a script of *ed* commands on specific lines. Note that the **g** command accepts as input all lines up to and including the first line that does not end in *e. Thus, the first

line that is not part of the **g** command above is the line containing **,p**. Note also that the period that usually must be typed to end the **i** command is not necessary if the line containing the period is also the last line of the **g** command. Thus, the period, along with the line on which it is typed, can be omitted.

A **g** command can be included in a **g** command list only when it is part of another command, as illustrated in the last example. It is illegal to try to nest command lists by specifying **g** command lists within other command lists.

The **v** command is identical to the **g** command, except that the command list is executed on all lines that do *not* contain the specified pattern.

If the results of a **g** command are not exactly what you had in mind, you can use the **u** command to restore your text to its previous state.

```
u
,p
The ed text editor is easy to use and easy to learn.
It was designed to enable the user to get his work done quickly
with the least possible amount of interference from the
editor. This is evident in the lack of prompts and the
curt error messages.
The ed text editor operates in two modes: command mode and
text entry mode. In command mode, the editor interprets
your input as a command. In text entry mode, ed adds
your input to the text located in a special buffer where
ed keeps a copy of the file you are editing. It is
important to note that ed always makes changes to the
copy of your file in the buffer. The contents of the
original file are not changed until you write the changes
on top of the original contents of your file.
It was designed to enable the user to get his work done
with the least possible amount of interference from the
editor. This is evident in the lack of prompts and the
curt error messages.
The ed text editor is easy to learn and easy to use.
The ed editor is great? The ed editor is great!
```

Note that the **u** command also reverses itself, so you can follow one **u** command with another to get back text that you have already reversed.

The **G** (interactive global) command is used when you have one command to execute on each line containing a specific pattern, but this command varies depending on the line. The **g** or **v** command is not appropriate in this case, since the command list for these commands is constant.

The **G** command is invoked in the form

x,*y***G**/*pattern*/

where *x* and *y* are line number arguments (if not specified, "1,$" is assumed), and *pattern* is the particular pattern you want to match in a line. **G** first marks every line containing a string that matches *pattern*. Then, dot is successively set to each marked line, and the resulting current line is printed on your screen. After the current line is printed, **G** waits for you to enter any single command, and the command you enter is executed. You may specify any command except the **a**, **i**, **c**, **g**, **G**, **v**, or **V** commands. Note that your command can address and affect lines other than the current line. A new-line is interpreted to be a null command. The **G** command can be terminated prematurely by pressing [DEL] or [BREAK]; otherwise it terminates normally when all lines in the file have been scanned for a string matching *pattern*.

Here is an example:

```
G/editor/
The ed text editor is easy to use and easy to learn.
s/easy/simple/
editor. This is evident in the lack of prompts and the

The ed text editor operates in two modes: command mode and
s/The ed text editor/ed/
text entry mode. In command mode, the editor interprets
s/the editor/ed/
editor. This is evident in the lack of prompts and the

The ed text editor is easy to learn and easy to use.
s/easy to use/simple to use/
The ed editor is great? The ed editor is great!
s/[^?]*? //
```

In this example, **G** looks for all the lines containing "editor", and executes the commands you specify. Note that a new-line was typed on each of the two blank lines above, causing no command to be executed.

The **&** character can be typed in place of a command. This causes the most recent command executed within the current invocation of **G** to be re-executed.

The **V** command is identical to the **G** command, except that the lines that are marked and printed are those that do *not* contain a string that matches *pattern.*

The **u** command can be used to reverse all the effects of a **G** command.

Joining Lines Together

The **j** (join) command joins two or more lines together. By default, **j** appends line dot+1 to line dot, but you can specify a range of lines to be joined. Note that **j** does not add any white space between the joined lines. Dot is left pointing to the line created after the specified lines have been joined.

As an example, try joining the last two lines of the file together. First, however, you need to shorten line $-1 so the joined line fits on one line of the screen. Do this by typing

```
$-1s/easy to learn and //p
The ed text editor is simple to use.
```

Now join the last two lines together with

```
jp
The ed text editor is simple to use.The ed editor is great!
s/\.T/. T/p
The ed text editor is simple to use. The ed editor is great!
```

The last **s** command in this example is used to insert two spaces between the two joined lines. Note that the **p** command can be appended to the **j** command to verify that the two lines have been joined.

Splitting Lines Apart

The **s** command can be used to split a single line into two separate lines. This is done by inserting a new-line between the characters where the split is desired. To do this, the new-line must be preceded by \ to avoid terminating the **s** command prematurely. Thus, you can split the two lines that were joined in the previous example into two separate lines with the **s** command (you cannot use the **u** command to split the last line into two lines now – why?). Do this by typing the following:

```
s/\. T/.T/p
The ed text editor is simple to use.The ed editor is great!
s/\.T/.\
T/
$-1,$p
The ed text editor is simple to use.
The ed editor is great!
```

The first **s** command gets rid of the extra white space in the sentence (note that the **u** command could have been used here). The second **s** command inserts a new-line between the period and the capital T, thus creating two separate lines. Note that, although the second **s** command takes up two lines, it is actually one command.

Special Ed Commands

Material Covered:

f	command; set/print currently remembered file name;
;	delimiter; set dot's value;
w	command; writer characters in buffer to file, or read standard output from a shell command;
r	command; read contents of file into buffer, or read standard output from shell command;
e, E	commands; begin editing another file, or read standard output from shell command;
-	option; silences character counts generated by **w**, **r**, **e**, **E**, or an invocation of *ed*;
X	command; initiates text encryption mode;
-x	option; initiates text encryption mode.

Finding the Currently Remembered File Name

If you invoke *ed* with a file name argument, *ed* remembers that file name until your editing session is over, or until the file name is changed as a result of commands that are discussed later in this section. The **f** (file name) command enables you to find out at any time what file name *ed* is remembering. For example,

```
f
testfile
```

which tells you that *ed* is remembering **testfile** as the current file name.

The **f** command also enables you to change the current file name. For example, to change the current file name to **file2**, type

```
f file2
file2
```

Ed echoes "file2" so you can verify that the current file is set correctly. Now change the file name back to the current file, or errors could result in later operations:

```
f testfile
testfile
```

If no file name is specified when *ed* is invoked, then *ed* initially remembers no current file name. Thus, this file name must be supplied when using the **w**, **r**, **e**, or **E** commands (discussed later), or it can be set with the **f** command.

Writing Buffer Text Onto a File

The **w** (write) command writes the text contained in the *ed* buffer onto the specified file, or onto the currently remembered file if no file name is specified. If the write is successful, a count of the number of characters written is printed. Dot is left unchanged.

The **w** command accepts zero, one, or two line number arguments specifying the line or lines to be written. If no line number arguments are given, "1,$" is assumed.

Try the **w** command by typing

```
w
986
```

The previous contents of **testfile** have now been overwritten by the contents of the *ed* buffer. The number 986 tells you that the write was successful, and that 986 characters were written.

Note that the *ed* buffer is not affected by the **w** command. Its contents are still the same. In fact, all of the line pointers (dot, **$**, and any that you have set) are still pointing to the same lines as they were prior to the **w** command. Thus, you may write out the contents of the *ed* buffer several times during an edit session without disturbing the current state of the editor. It is a good idea to write often, especially if you have been editing a long time and have made many changes. Depending on how often you write, you can be sure that a current version of your file resides in the relative safety of the file system, should a system crash or a power failure eat up whatever data is in the *ed* buffer.

You can tell *ed* to write to a file other than the currently remembered file by typing

```
/^ed/;/^on/w file1
561
```

This command writes the range of lines from the line beginning with "ed" to the line beginning with "on" onto the file **file1**. If **file1** exists, its previous contents are completely overwritten by the specified lines of text. If **file1** does not exist, it is created with a file mode of 666 (modified by the current value of the file creation mask, *umask*) and the specified text is written on it. Again, the number returned indicates that *ed* was successful in writing 561 characters on the file.

The semicolon that appears in the last example is new. If a comma had been used to separate the two searches, *ed* would have started the search for a line beginning with "ed" from the current line. After finding that line, however, *ed* would return to the current line to search for the line beginning with "on". The value of dot would be reset only after finding the line beginning with "on", with the result that a single line address is passed to the **w** command, causing a single line to be written. The semicolon causes the value of dot to be set to the line beginning with "ed", so that the second search is carried out with respect to this line, instead of the previous current line. Thus, two addresses are processed, and the correct lines are written. The semicolon can always be used in place of a comma to force dot to be set at that point in the construct.

You can also run shell commands with the **w** command. The shell command is introduced with **!**. For example,

```
w !ls
file1
testfile
986
```

runs *ls* and also writes the current contents of the buffer to the currently remembered file. Note that the output from *ls* appears on your screen, but is not added to the actual contents of the buffer (the listing that appears on your screen may be longer than that shown above). After the listing is produced, *ed* writes the contents of your buffer to the currently remembered file, and reports the number of characters written. Note that there is no way to run a shell command and write to a file other than the currently remembered file with the **w** command. Note also that **!** is illegal if the editor was invoked from a restricted shell (see *rsh*(1) in the *HP-UX Reference manual*).

The currently remembered file name is set to the file name you specify with the **w** command, if the specified file name is the first file name mentioned since *ed* was invoked. Otherwise, the currently remembered file name is not affected. A shell command introduced with **!** is never remembered as the current file name.

Reading Files Into the Buffer

The **r** (read) command reads the contents of a specified file, or the currently remembered file, if no file is specified, into the *ed* buffer after the specified line. If no line is specified, the contents are read in after line $. Dot is set to the last line read in.

To illustrate the **r** command, first create a new file called **readfile**:

```
w
986
e readfile
?readfile
a
Here is some text that is to be read in.
It is used to illustrate the r command.
.
w
81
```

You now have a file in your working directory called **readfile**, containing the text shown above. Now begin editing **testfile** again, and read in the contents of **readfile**:

```
e testfile
986
0r readfile
81
1,5p
Here is some text that is to be read in.
It is used to illustrate the r command.
The ed text editor is simple to use and easy to learn.
It was designed to enable the user to get his work done quickly
with the least possible amount of interference from the
```

This example reads the contents of **readfile** into **testfile** after line 0, or at the beginning of the file. *Ed* responds by printing the number of characters that were read in. The first five lines of the buffer are printed to verify that the text is placed correctly.

You can also run shell commands with the **r** command. The shell command is introduced with **!**. For example,

```
/curt/r !date
29
6,9p
editor. This is evident in the lack of prompts and the
curt error messages.
Thu Jul 22 10:59:13 MDT 1982
ed operates in two modes: command mode and
```

which reads the output from *date* into **testfile** after the line containing the pattern "curt". The lines surrounding the insertion are printed to verify that the read executed correctly. Note that, unlike the **w** command, the output from the command becomes part of the text in the buffer. Also, the number of characters read from the command is printed on your screen, but the actual output appears only in the buffer. Note that the **!** is illegal if the editor was invoked from a restricted shell.

The currently remembered file name is reset to the file name you specify with the **r** command, if the specified file name is the first file name mentioned since *ed* was invoked. Otherwise, the currently remembered file name is not affected. A shell command introduced by **!** is never remembered as the current file name.

An **r** command can be reversed with the **u** command. Try this now:

```
u
6,8p
editor. This is evident in the lack of prompts and the
curt error messages.
ed operates in two modes: command mode and
```

Note that the date and time are no longer present in the buffer.

Editing Other Files

The **e** (edit) command discards the entire contents of the *ed* buffer and reads in the specified file. If no file is specified, then the currently remembered file is read. Dot is set to the last line of the buffer.

If you have made any changes to the buffer since the last **w** command, *ed* requires that you precede the **e** command with a **w** command to save the contents of the buffer. If you are sure that you want to discard the contents of the buffer, you can invoke the **e** command a second time. This forces *ed* to discard the buffer contents and read in the new file. For example,

```
e file1
?
e file1
561
```

The question mark after the first invocation of **e** is to warn you that you have made changes to the current contents of the buffer, and that these changes will be lost if you do not write them on **testfile**. The second invocation of **e** tells *ed* "I don't care! Do it anyway!". *Ed* complies by discarding the current buffer and reading in the contents of **file1**. *Ed* reports to you the number of characters read.

If you are sure that you want to discard the current contents of the buffer without saving them, you can use the **E** (Edit) command. **E** is similar to **e**, except that *ed* does not check to see if any changes have been made to the current buffer. Thus, you do not have to type the **e** command twice.

If you have made several changes to the buffer, and then decide that you do not like what you have done, you can start editing the same file all over again by typing **e** or **E** with no specified file name. This causes the contents of the currently remembered file to be read into the buffer, destroying the previous contents. Of course, if you have written some of the changes you have made to the current file already, there is no quick and easy way to reverse them.

If you specify a file name with the **e** or **E** command, that file name becomes the new current file, and is remembered for future use with **w**, **r**, **e**, or **E**.

You can also execute shell commands with the **e** or **E** command. The shell command is introduced with **!**. For example,

```
E !ls
23
,p
file1
readfile
testfile
```

This example runs the shell command *ls*, and places its output in the *ed* buffer, destroying whatever was in the buffer previously. The number of characters placed in the buffer is printed for you. The actual list of files and the number of characters read into the buffer may be different than those shown above. Note that **!** is illegal if the editor was invoked from a restricted shell A shell command is never remembered as the current file name.

Silencing the Character Counts

If the character counts that *ed* produces (when *ed* is invoked, or with the **w**, **r**, **e**, or **E** commands) are annoying or are not helpful, they can be silenced with the - option. It is specified when *ed* is invoked, as in

```
$ ed - filename
```

The - option also suppresses the question mark generated by the **e** and **q** commands whenever they are not preceded by a **w** command (the **q** command is discussed in the next section).

Encrypting and Decrypting Text

Ed provides a feature that enables you to encrypt and decrypt the text in a file so that other users are not able to read your files. The text is encrypted and decrypted by means of the DES encryption algorithm (see *crypt*(1) in the HP-UX Reference manual). To encrypt your text, you must supply a *key*, which is simply a string of one or more characters. The key determines the manner in which the DES algorithm encrypts your text. You *must* remember this key.

The **X** (encrypt) command enables you to encrypt the text in the *ed* buffer. The **X** command accepts no arguments, but prompts you to enter a key. The echoing on your screen is disabled while you enter the key, so there is no visible record of it. For example,

```
E file1
561
,p
editor. This is evident in the lack of prompts and the
curt error messages.
The ed text editor operates in two modes: command mode and
text entry mode. In command mode, ed interprets
your input as a command. In text entry mode, ed adds
your input to the text located in a special buffer where
ed keeps a copy of the file you are editing. It is
important to note that ed always makes changes to the
copy of your file in the buffer. The contents of the
original file are not changed until you write the changes
on top of the original contents of your file.
X
Enter file encryption key:
w
561
q
$
```

This example edits **file1**, and prints out its contents. After the **X** command is invoked, you are prompted to enter a key. This key can be any string of characters, but whatever it is, *do not forget your key*! When the **w** command is invoked, the text in the buffer is encrypted according to the key you entered and written on **file1**. The **q** command, which is discussed later, exits the editor and leaves you at the shell level. Now execute the *cat* command to try to print out the contents of **file1**:

```
$ cat file1
    (garbage)
    .
    .
    .
$
```

You probably got a screenful of garbage. If your bell beeped a couple of times, this is because the text is encrypted into invisible characters as well as visible characters. There is no practical way for another user to tell what is actually contained in your file.

To edit a file containing encrypted text, use the **-x** option when *ed* is invoked:

```
$ ed -x file1
Enter file encryption key:
561
,p
editor. This is evident in the lack of prompts and the
curt error messages.
The ed text editor operates in two modes: command mode and
text entry mode. In command mode, ed interprets
your input as a command. In text entry mode, ed adds
your input to the text located in a special buffer where
ed keeps a copy of the file you are editing. It is
important to note that ed always makes changes to the
copy of your file in the buffer. The contents of the
original file are not changed until you write the changes
on top of the original contents of your file.
```

The **-x** option is the same as the **X** command, except that it is used when you invoke *ed*. When prompted for the key, you must enter the same key that you entered when the text was encrypted. Otherwise, the text in that file is inaccessible. This is why it is so important that you remember your key. After the key is entered, the text in **file1** is decrypted and read into the *ed* buffer. You may now edit the text normally.

When you are done editing, if you invoke the **w** command to write your changes to the file, the text is encrypted according to your key. If you want to change your key or disable encryption altogether, you must use the **X** command. When you are prompted for your key, either type in your new key to change the encryption key, or simply type a new-line. If you type a new-line, a null key is entered, and encryption is disabled. Disable encryption now by typing

```
X
Enter file encryption key: (new-line)
w
561
```

The contents of **file1** are now in a readable form.

Note that, when encryption is enabled, all subsequent **e**, **r**, and **w** commands encrypt the text in the *ed* buffer.

As a general rule, text encryption is seldom needed by the typical user except when extreme security is required. The HP-UX file system has its own security system which is sufficient for most security needs. Using text encryption often and/or on several files at once is a dangerous practice, since you must remember your key to successfully edit these files. You should therefore exercise caution when using the text encryption feature.

The Shell Interface

Material Covered:

!	command; execute shell command;
q	command; exit editor after checking for changes to the buffer;
Q	command; exit editor without checking buffer for changes.

Escaping to the Shell Temporarily

The **!** command enables you to execute a shell command from within the *ed* editor. To do this, type a **!**, followed by the shell command. For example,

```
!(date;who) > whofile
!
```

executes the *date* and *who* commands, and redirects their output into the file **whofile**. Note that *ed* returns a **!** to tell you when the command has completed execution.

If the character **%** appears anywhere in the shell command, it is replaced with the currently remembered file name. Thus,

```
!sort % > sortedfile
sort file1 > sortedfile
!
```

sorts (in reverse alphabetical order) the current contents of **file1**. Note that the current contents of **file1**, not the *ed* buffer, are sorted. The sorted version of **file1** is redirected to the file **sortedfile**. The I/O redirection in the last two examples is used so that the output from these shell commands does not clutter up your screen while you are editing. Note that, if the output from a shell command is printed on your screen, the output does not become part of the *ed* buffer unless **!** is used with the **r**, **e**, or **E** commands.

A final feature of the ! command is the ability to re-execute the last shell command you executed with !, without having to retype the entire command. This is done by typing two exclamation points, as in

```
!!
!
```

which re-executes the last shell command executed within the *ed* editor. Thus, **sort % >sortedfile** is re-executed.

If a shell command contains any metacharacters, *ed* echoes the command line back to you with all metacharacters expanded (this is what *ed* did in the first *sort* example above). For example,

```
!cat * > bigfile
cat file1 readfile sortedfile testfile whofile >bigfile
!
```

which echoes the expanded command line, then executes the command.

Exiting the Editor

The **q** (quit) command exits the editor. The contents of the buffer are not automatically written on the current file. If you have made any changes to the buffer since the last time you invoked the **w** command, *ed* requires that you issue the **w** command before exiting with **q**. Invoking the **q** command a second time forces *ed* to let you exit without writing the contents of the buffer on the current file. To illustrate this command, first add some text to the buffer, then try to exit without writing:

```
$a
Here is some extra text.
.
q
?
q
$
```

A change is made to the buffer by adding a single line of text to the end of the buffer. When the first **q** command is typed, *ed* sees that there have been changes to the buffer since the last write, so *ed* issues a question mark. This warns you that there are changes to the text in the buffer that will not be saved if you exit without writing. The second **q** command forces *ed* to discard the contents of the buffer and exit. Be very sure that this is what you want to do, since you cannot recover the buffer contents once you have

exited. The $ is the default shell prompt, indicating that you are once more at the shell level (your shell prompt may be different).

If you know that you want to discard the contents of the buffer and exit, but you do not want to type the **q** command twice, use the **Q** command. The **Q** command is similar to **q**, but *ed* does not check to see if changes have been made to the contents of the buffer.

The - option previously discussed disables the question mark that *ed* issues when you do not write before executing an **e** or **q** command. You are living dangerously when it is disabled, however. That question mark has kept many users from accidentally throwing away hours of work. Besides, the **E** and **Q** commands are implemented for those special cases when you want to discard the contents of the buffer.

Miscellaneous Topics

Material Covered:

[DEL],[RUB],[BREAK] keys; any of these keys generates an interrupt signal to *ed*;

Editing Scripts

Interrupting the Editor

[DEL], [RUB], or [BREAK] causes *ed* to stop whatever command it is executing and return to you for a command. *Ed* tries to restore the state of your file to whatever it was before the command was issued. This is easily done if *ed* is interrupted while printing, since dot is not set until printing is done. If *ed* is reading or writing files, or performing substitutions or deletions, however, the state of the buffer (and the current file) is unpredictable; dot may or may not be changed. Thus, it is usually safer to let *ed* finish whatever it is doing, rather than risk finding the buffer or the current file in some garbled state.

Editing Scripts

An editing script is simply a file containing a list of *ed* commands. If you have several files on which a specific list of commands must be executed, it is easier to use an editing script than it is to invoke *ed* once for every file, and perform the tasks in each.

Suppose you have several files named **file1**, **file2**, ..., and you want to perform some specific substitutions, additions, and deletions on each. First, create a file (called **script**, for example), and put all the *ed* commands that you want to execute, in the order that they must be executed, in the file:

```
$ ed script
?script
a
0r !date
1s/.*$/& DATE OF LAST UPDATE/
$-3,$d
g/Karl Harrison/s//Georgia Mitchell/
w
q
.
w
87
q
$
```

The file **script** now contains *ed* commands to put the current date and time at the beginning of each file, append "DATE OF LAST UPDATE" to the date and time, delete the last four lines of each file, and replace every instance of "Karl Harrison" in each file with "Georgia Mitchell". Note that the **w** and **q** commands are included so that the script writes the buffer on each file and exits the editor automatically.

To use **script**, invoke *ed* as follows:

```
$ ed - file1 <script
$ ed - file2 <script
        etc.
```

The I/O redirection character < causes *ed*, when invoked, to take its input from **script**. Thus, as *ed* is invoked with each file name, that file is edited according to the commands contained in **script**.

Index

a

b

c

d

e

f

g

i

j

l

m

p

r

s

t

u

Table of Contents

EDIT – An Interactive Line Editor 1

Introduction

The editor program *edit* is a somewhat simplified version of the *ex* editor described earlier in this volume. It is most useful on older-model typewriter terminals and generally is of little interest to users who have access to the intelligent CRT display termials that are most commonly used with the HP-UX operating system.

This tutorial is divided into four lessons and assumes no prior familiarity with computers or with text editing. A series of text editing sessions lead you through the basic steps of creating and revising a text file. After scanning each lesson but before beginning the next, try the examples at a terminal to get a feeling for the actual process of text editing. Allow time for experimentation, and you can quickly learn to use a computer for writing and modifying text.

Other HP-UX features are useful besides the text editor. These features are discussed in the book *Introducing UNIX System V* as well as in other tutorials that provide a general introduction to the system. As soon as you are familiar with your terminal keyboard, its special keys, the system login procedure, and know how to correct typing errors, you are ready to start. Let's first define some terms:

Program A group of computer instructions that defines the sequence of steps to be performed by the computer in order to accomplish a specific task. For example, a series of steps to balance your checkbook is a program.

HP-UX A special type of program called an operating system that supervises the computer, peripheral devices, and all programs that use the HP-UX operating system.

Edit The name of the HP-UX text editor that you will be learning to use; a program that aids you in writing or revising text. *Edit* was designed for beginning users, and is a simplified version of a more extensive editor named *ex*.

File Each HP-UX account is allotted disk storage space for permanent storage of information such as programs, data, or text. A file is a logical collection of data (such as an essay, a program, or a chapter from a book) that is stored and maintained by a computer system. Once you create a file it is kept until you tell the system to remove it. You can create a file during one HP-UX session, log out, then return to use it at a later time. Files contain anything you choose to write and store in them. File sizes vary, depending on individual needs. One file might contain a single number, while another could contain a very long document or program. The only way to save information from one session to the next is to store it in a file where it is kept for later use.

Filename Filenames are used to distinguish one file from another, serving the same purpose as labels on manila folders in a file cabinet. To write or access information in a file, use the name of that file in an HP-UX command. The system automatically determines where the file is located.

Disk Files are stored on a thin circular disk that is coated with magnetic particles similar to magnetic recording tape. The disk may be permanently installed in a disk drive, or it may be a removable flexible disk that resembes a small phonograph record in a thin, square protective container or envelope. Information from the computer (such as your text) is recorded on the disk surface by the disk drive.

Buffer A temporary work space that is available to the user during a text editing session. The buffer is used to build and modify text files. Buffers are analagous to a piece of scratch paper that is diskarded at the end of a session after the information it contained has been copied (written) to a permanent disk file.

Session 1
Creating a Text File

Before you can use the editor, you must first log onto the computer so HP-UX can set up communication between your terminal and the editor program. Here is a review of the standard HP-UX login procedure:

If the terminal you are using is directly linked to the computer, turn it on and press [Return] (or [Enter]). If your terminal uses an acoustic coupler (or modem) and telephone line instead, turn on the terminal, dial the system-access telephone number, then, when you hear a high-pitched tone, place the telephone receiver in the acoustic coupler. If you are using a modem, consult the modem manual for procedures. Press carriage return (or the [Return] key) once, and await the login message:

```
:login:
```

Type your login name (which identifies you to HP-UX) on the same line as the login message, then press [Return]. If the keyboard on your terminal supports both uppercase and lowercase, be sure you enter your login name in lowercase. Otherwise, HP-UX assumes your terminal has only uppercase and will not recognize any lowercase letters you may type. When HP-UX types `:login:`, reply with your login name, for example **susan**:

```
:login: susan [Return]
```

(In this example, input typed by the user appears in bold face to distinguish it from information displayed by HP-UX.)

HP-UX responds with a request for a password as an additional precaution to prevent unauthorized people from using your account. The password will not appear when you type it (to prevent others from seeing it). The message is:

```
Password: _
```
(type your password and press [Return])

If any of the information you gave during the login sequence was mistyped or incorrect, HP-UX responds with:

```
Login incorrect.
:login:
```

If this happens, start over and repeat the process. When you successfully log in, HP-UX prints the message of the day and eventually presents you with a % at the beginning of a fresh line. The % is the HP-UX prompt symbol that tells you HP-UX is ready to accept a command.

Asking for *edit*

You are ready to tell HP-UX that you want to use *edit*, the text editor program. Now is a convenient time to choose a name for the text file you are about to create. To begin your editing session type `edit` followed by a space, then the filename you have selected, such as *text*. When you have completed the command, press Return and wait for *edit*'s response:

```
% edit text
"text" no such file or directory
:
```

If you typed the command correctly, you will now be in communication with *edit*. *Edit* has set aside a buffer for use as a temporary working space during your current editing session. It also checked to see if the file you named, *text*, already exists. As we expected, it was unable to find such a file since *text* is the name of the new file to be created. *Edit* confirms this with the line:

```
"text" No such file or directory
```

The colon on the next line is *edit*'s prompt, announcing that *edit* expects a command from you. You are now ready to create the new file.

The "Command not found" Message

Suppose you misspelled *edit* by typing *editor*. Your request would be handled as follows:

```
% editor
editor:  Command not found.
%
```

Your mistake in calling *edit* `editor` was treated by HP-UX as a request for a program named *editor*. Since there is no program named *editor*, HP-UX reported that the program or command could not be found. A new % prompt indicates that HP-UX is ready for another command, so you can now enter the correct command.

Summary

Your exchange with HP-UX as you logged in and made contact with edit should look something like this:

```
:login: susan
Password:
   ... A message of General Interest...
% edit text
"text" No such file or directory
:
```

Creating text

You can now begin to enter text into the buffer. This is done by appending text to whatever is currently in the buffer. Since there is nothing in the buffer at the moment, you are appending text to nothing which, in effect, creates text. Most edit commands have two forms: a word that describes what the command does and a shorter abbreviation of that word. Either form can be used. Many beginners find the full command names easier to remember, but once you are familiar with editing you may prefer to type the shorter abbreviations. The command to input text is *append* which can be abbreviated *a*. Type **append**, then press Return.

```
% edit text
:append
```

Messages from *edit*

If you make a mistake while entering a command and type something that *edit* does not recognize, *edit* responds with a message intended to help you diagnose your error. For example, if you misspell the command to input text by typing perhaps, *add* instead of *append* or *a*, you receive this message:

```
:add
add:Not an editor command
:
```

When you receive a diagnostic message, examine what you typed to determine what part of your command confused *edit*. The message above means that *edit* could not recognize your mistyped command, so the command was ignored. After displaying a new colon prompt, *edit* is now ready to receive a new command.

Text Input Mode

By giving the command *append* (or using the abbreviation *a*), you activated **text input mode**, also known as **append mode**. When you enter text input mode, *edit* responds by doing nothing. No prompts appear during text input mode, your signal to begin entering lines of text. You can type almost anything you want while inputting text lines. Lines are transmitted one at a time to the buffer and held there during the editing session. You can append as much text as you want. When you are through entering new text lines, type a period by itself at the beginning of a new line, then press [Return]. This signals the editor to terminate text input mode and return to **command mode**. *Edit* then prompts you for a new command by displaying a colon (:) prompt.

When you leave *append* mode and return to command mode (necessary in order to do any of the other kinds of editing, such as changing, adding, or printing text), *edit* preserves the text you just typed in the editor buffer, so nothing is lost. If you type any other character besides a period by itself on the last line, *edit* treats the line as text instead of an exit command, and will not let you leave *append*. To exit, type a period by itself on a single line terminated by [Return].

This is a good place to learn an important lesson about computers and text: **as far as the computer is concerned, a blank space is a character as distinct as any letter of the alphabet. If you so much as type a blank after the period** (that is, type a period then press the space bar on the keyboard), **you will remain in append mode** with the last line of text being a period followed by a single space.

Let's say that the lines of text you enter are (try to type exactly what you see, including "thiss"):

```
This is some sample text.
And thiss is some more text.
Text editing is strange, but nice.
```

The last line is the period followed by [Return] that gets you out of append mode. If, while typing the line, you hit an incorrect character, you can change the incorrect character by using the [Back space] key to back up then retype the line beginning with the incorrect character. If you back-space to the first character in the line then press [Return], a blank line is stored in the buffer. Corrections to a line must be done before the line has been completed by a [Return] (changes in lines already typed are discussed in Session 2).

Writing Text to Disk

Text input is now complete. Before you break for lunch, the text should be put in a disk file for safekeeping until the next editing session. Storing the editor's buffer in a disk file is the only way to save information from one session to the next, since the buffer is temporary and is destroyed after the end of the editing session. Thus, learning how to write a file to disk is second in importance only to entering the text. To write the contents of the buffer to a disk file, use the command, *write* (or its abbreviation *w*):

```
:write
```

Edit now copies the buffer to a disk file. If the file does not exist, a new file is created automatically and the presence of a "New File" will be noted. The newly-created file is given the name specified when you entered the editor, in this case, *text*. To confirm that the disk file has been successfully written, *edit* repeats the filename, then gives the number of lines and the total number of characters in the file. The buffer remains unchanged by the *write* command. All of the lines that were written to the disk are still in the buffer, should you want to modify or add to them.

This ability to write a file to the disk and still continue editing is useful insurance against loss of data during power failures. It is a good idea to periodically write the edit buffer to a permanent file to minimize the risk of losing an hour's work should the power go off for some reason (the risk is actually not as serious as this sounds, because HP-UX has recovery commands that recover all but very little of the file in the event of a power failure).

Edit must have a filename to use before it can write a file. If you forgot to indicate the name of the file when you began the editing session, *edit* prints:

```
No current filename
```

in response to your *write* command. If this happens, you can specify the filename in a new *write* command:

```
:write text
```

After the *write* (or *w*) type a space followed by the name of the file.

Logging Off

We have done enough for this first lesson on using the HP-UX text editor, and are ready to terminate (quit) the editing session. To do this, type *quit* (or *q*), then press Return. The terminal display looks like this:

```
:write
"text" [New file] 3 lines, 90 characters
:quit
%
```

The (%) prompt is from HP-UX, telling you that your session with *edit* is over and you can now interact with HP-UX. To end the entire session at the terminal, you must also exit from HP-UX. In response to the HP-UX prompt of "%" press the CTRL and D keys simultaneously to terminate the session with HP-UX and make the terminal available to the next user. It is always important to logout at the end of a session to make absolutely sure no one could accidentally stumble into your abandoned session and thus gain access to your files, a condition that tempts even the most honest of souls.

This is the end of the first session on HP-UX text editing.

Session 2

Login with HP-UX as in the first session:

```
:login:susan [Return]
Password:      (give password then press [Return])
%
```

This time when you type the *edit* command, you can specify the name of the file you worked on last time. Thus, when *edit* starts, it will transfer the original file into its buffer so that you can resume editing the same file. When edit has copied the file into the buffer, it shows the original file name, and lists the number of lines and characters in the file as follows:

```
%edit text
"text" 3 lines, 90 characters
:
```

Your command to edit file *text* caused the editor to copy the 90 characters of text into the buffer. *Edit* now awaits your next command. In this session you learn to append more text to the file, print the contents of the buffer, and change the text in a line.

Adding More Text to the File

To add more to the end of your text, use the *append* command to enter text input mode. When *append* is the first command of your editing session, the lines you enter are placed at the end of the buffer. Why this happens is explained later in this session. This time, use the abbreviation for the *append* command: *a*:

```
:a
This is text added in Session 2.
It doesn't mean much here, but
it does illustrate the editor.
.
```

Interrupt

Most terminals supported by HP-UX have a [DEL] (delete) key. If you press [DEL] while working with *edit*, any task the editor is performing is stopped, and the following message is sent to you:

```
Interrupt
:
```

Any command that edit might be executing is terminated by [DEL], causing *edit* to prompt you for a new command. If you are appending text at the time the key is pressed, append mode terminates and you are expected to give another command. The line of text that you were typing when the *append* operation was interrupted is lost and is not entered into the buffer.

Making Corrections

If you have read a general HP-UX introductory text, you will recall that it is possible to erase individual letters that you have typed. This is done by typing the designated erase character as many times as there are characters you want to erase. Accounts normally start out using the number sign (#) as the erase character, but it's possible for a different erase character to be selected. We'll show "#" as the erase character in our examples, but if you've changed your erase character to backspace (control-H) or something else, be sure to use your own erase character.

If you make a bad start in a line and would like to begin again, erasing individual characters with a "#" is cumbersome – what if you had 15 characters in your line and wanted to get rid of them? To do so either requires:

```
This is yukky tex###############
```

with no room for the great text you would like to type, or,

```
This is yukky tex@This is great text.
```

When you type the at-sign (@), you erase the entire line typed so far. (An account can select a different line erase character to use in place of @. If your line-erase character has been changed, use it where the examples show "@",). You can immediately begin to retype the line. This, unfortunately, does not help after you type the line and press [Return]. To make corrections in completed lines, it is necessary to use the editing commands covered in this and following sessions.

HP-UX and *edit* also support use of [Back space] for text corrections. How the backspace key affects the terminal screen display depends on how your terminal or terminal emulator functions. You can look it up in the manual, or just try it out.

Listing Buffer Contents

Having appended text to what you wrote in Session 1, you might be curious to see what is in the buffer. To print the contents of the buffer, type the command:

```
:1,$p
```

The "1" stands for line 1 of the buffer, the "$" is a special symbol designating the last line of the buffer, and *p* (for *print*) is the command to print from line 1 to the end of the Thus, `1,$p` gives you:

```
This is some sample text.
And thiss is some more text.
Text editing is strange, but nice.
This  is  text  added in Session 2.  It doesn't mean much
It doesn't mean much here, but
it does illustrate the editor.
```

You may occasionally place a character in the buffer that cannot be printed (ASCII control characters are not printed on most output devices). These characters are usually obtained by pressing CTRL and some other key at the same time. When printing lines, *edit* uses a special notation to show the existence of non-printing (control) characters.

Suppose you had introduced the non-printing character "control-A" into the word "illustrate" by accidently holding down the CTRL key while typing **a**. If you asked to have the line printed, **EDIT** would display:

```
it does illustr'Ate the editor.
```

The two-character sequence `'A` indicates that the CTRL key was depressed simultaneously with the "A" key, resulting in a corresponding control character (the apostrophe indicates that CTRL was pressed). The error is easily corrected, as discussed later in this session.

In looking over the text we see that "this" is typed as "thiss" in the second line, as was previously suggested. Let's correct the spelling.

Finding Things in the Buffer

You must find something in the buffer before you can change it. To find "thiss" in the text you entered, look at a listing of the lines. *Edit* searches the buffer, looking for the text sequence "thiss", and stops searching when it finds the specified character pattern. You can tell *edit* to search for a pattern by typing the pattern between slash marks:

```
:/thiss/
```

By typing `/thiss/` and pressing [Return], *edit* is instructed to search for "thiss" (if *edit* cannot find the pattern of characters in the buffer, it responds "Pattern not found"). When edit finds the characters "thiss", it prints the line where the pattern was found for your inspection:

```
And thiss is some more text.
```

Edit is now positioned in the buffer at the line which it just printed, ready to make a change in the line.

The Current Line

Edit always keeps track of its position in the buffer by identifying the "current line" at the end of each operation. In general, the line that was most recently printed, entered, or changed is considered to be the current position or line in the buffer. The editor assumes the next command is to be applied to the current line, unless you direct it to act in another location (or perform an operation that is not related to the current line). When you bring a file into the editor, the editor is always positioned at the last line in the file. If your initial editing command is "append", the lines you enter are added to the end of the file, that is, they are placed after the current line. You can refer to your current postion in the buffer by the symbol period (.), usually called "dot". If you type "." then press [Return], you are telling *edit* to print the current line:

```
:.
And thiss is some more text.
```

If you want to know the number of the current line, you can type .= and carriage return, and edit will respond with the line number:

```
:.=
2
```

If you type the number of any line and a carriage return, edit will position you at that line and print its contents:

```
:2
And thiss is some more text.
```

Experiment with these commands to ensure that you understand what they do.

Numbering Lines (nu)

The *number* (*nu*) command is similar to *print*, giving both the number and the text of each printed line. To see the number and text of the current line, type

```
:nu
2  And thiss is some more text.
```

Notice that the shortest abbreviation for the *number* command is *nu* (not *n* which is used for a different command). You can specify a range of lines to be listed by the number command in the same way that lines are specified for print. For example, `1,$nu` lists all lines in the buffer and their corresponding line numbers.

Substitute Command (s)

Now that you have found the misspelled word, it is time to change "thiss" to "this". As far as *edit* is concerned, changing text is a matter of substituting one pattern for another. Just as *a* stood for *append*, so *s* stands for *substitute*. Use the abbreviation *s* to reduce the chance of mistyping the *substitute* command. This command instructs *edit* to make the change:

```
2s/thiss/this/
```

First, indicate the line to be changed (2), then type the command (*s*), followed by the characters to be removed (typed between slashes). Finish the line with the characters to be put back in followed by a closing slash mark, then press Return. Here it is in plain English:

```
2s/what is to be changed/what to change to/
```

If *edit* finds an exact match of the characters to be changed it makes the change **only** in the first occurrence of the characters. If it does not find the characters to be changed it will respond:

```
Substitute pattern match failed
```

indicating that your instructions could not be carried out. If *edit* finds the characters you want to change, it makes the sutstitution and automatically prints the changed line so you can verify that the correct substitution was made. In the example,

```
:2s/thiss/this/
And this is some more text.
:
```

line 2 (and line 2 only) is searched for the character pattern "thiss". When the first exact match is found, "thiss" is changed to "this". In reality, since you set the current line number to 2 in an earlier operation, it was unnecessary to specify the number of the line to be changed by this command. In the command:

```
:s/thiss/this/
```

edit assumes that the line where the editor is currently positioned (the **current line**) is to be used. A period can also be used to specify the current line as in the command:

```
:.s/thiss/this/
```

although the period is totally unnecessary. In either case, the command without a line number or without a period would have produced the same result as when the line number was specified because the editor was already positioned at the line to be changed. Here is another illustration of substitution.

```
Text editing is strange, but nice.
```

To be a bit more positive, take out the characters "strange, but" so the line reads:

```
Text editing is nice.
```

A command that positions *edit* at that line then makes the substitution is:

```
:/strange/s/strange, but //
```

This command combines the search with a substitution, a perfectly allowable combination. Thus, you do not necessarily have to use line numbers to identify a line to edit. Instead, you can identify the line to be changed by asking *edit* to search for a specified pattern of characters that occurs in the line of interest. The function of each part of the command is as follows:

`/strange/`	tells edit to find the characters "strange" in the text
`s`	tells edit we want to make a substitution
`/strange,but //`	substitutes nothing at all for the characters "strange, but "

Note the space after "but" on "/strange, but /". If you do not indicate the space is to be taken out, your line becomes:

```
Text editing is  nice.
```

which looks odd because of the extra space between "is" and "nice". Again, you can see that a blank space is a real character to a computer, and when editing text you need to be aware of spaces within a line just as you would be aware of an "a" or a "4".

Another Way to List What's in the Buffer (z)

Although the *print* command is useful for looking at specific lines in the buffer, other commands can be more conveniet for viewing large sections of text. You can ask to see a screen full of text at a time by using the command *z*. If you type

`:1z` Return

edit starts with line 1 and continues printing lines, stopping either when the screen of your terminal is full, or when the last line in the buffer has been printed. If you want to read the next segment of text, type the command

`:z` Return

If no starting line number is given for the *z* command, printing starts at the "current" line; in this case the last line printed. Viewing lines in the buffer one full screen at a time is known as paging. Paging can also be used to print a section of text on a printing terminal.

Saving the Modified Text

Now is a good place to pause and end the second session. If you hastily type **q** Return to terminate the session, your interaction with *edit* resembles:

```
:q
No write since last change (q! quits)
:
```

This is *edit*'s warning that you have not written the modified contents of the buffer to disk. You are risking the loss of the work you have done during the editing session since the last previous *write* command. Since no previous disk write was performed during this session, everything done during the session would be lost. If you do not want to save the work done during this editing session, you can type *q!* to confirm that you indeed want to end the session immediately, losing the contents of the buffer. However, since you probably prefer to preserve the edited file, use the *write* command as follows:

```
:w
"text" 6 lines, 171 characters
```

then follow with

```
:q
%logout
```

and hang up the phone or turn off the terminal when HP-UX asks for a login name.

This is the end of the second session on HP-UX text editing.

Session 3

Bringing Text Into the Buffer (e)

Login to UNIX and make contact with *edit*. Try to do it without using notes if you can.

Did you remember to give the name of the file you wanted to edit by typing:

```
%edit text
```

or did you type:

```
%edit
```

Both commands activate *edit*, but only the first version can bring a copy of the file named *text* into the buffer. If you forgot to specify the filename, you can recover by typing:

```
:e text
"text" 6 lines, 171 characters
```

The *edit* command which can be abbreviated *e* when you're in the editor, tells *edit* that you want to destroy anything already in the editor's buffer and copy the file *text* into the buffer for editing. You can also use the *edit* (*e*) command to change files in the middle of an editing session or to give *edit* the name of a new file that you want to create. Because the *edit* command clears the buffer, you will receive a warning if you try to edit a new file without having saved a copy of the old file. This gives you a chance to write the contents of the buffer to disk before editing the next file.

Moving Text in the Buffer (m)

Edit enables you to move lines of text from one location in the buffer to another by means of the *move* (*m*) command:

```
:2,4m$
```

This example directs *edit* to move lines 2, 3, and 4 to the end of the buffer following the last line, indicated by ($). When constructing the *move* command, specify the first line to be moved, the last line to be moved, the move command *m*, then the line after which the moved text is to be placed. Thus,

```
:1,6m20
```

commands *edit* to move lines 1 through 6 (inclusive) to a position immediately following line 20 in the buffer. To move only one line, say line 4, to a position in the buffer after line 6, the command would be "4m6".

Let's move some text using the command:

```
:5,$m1
2 lines moved
it does illustrate the editor.
```

After executing a command that changes more than one line of the buffer, *edit* tells how many lines were affected by the change. The last moved line is printed for your inspection. If you want to see more than just the last line, use the print (p), z, or number (nu) command to view more text. The buffer should now contain:

```
This is some sample text.
It doesn't mean much here, but
it does illustrate the editor.
And this is some more text.
Text editing is nice.
This is text added in Session 2.
```

You can restore the original order by typing:

```
:4,$m1
```

or you can combine context searching and the *move* command for the same result:

```
:/And this is some/,/This is text/m/This is some sample/
```

The danger in combining context searching with the move command lies in the higher probability of making a typing error in such a long command. Typing line numbers is usually much safer.

Copying Lines (copy)

The *copy* command is used to make a second copy of specified lines. leaving the original lines where they were. *Copy* has the same format as the *move* command. For example:

```
:12,15copy$
```

makes a copy of lines 12 through 15, placing the added lines after the last line in the buffer ($). Experiment with the `copy` command so that you can become familiar with how it works. Note that the shortest abbreviation for *copy* is *co* (and not the letter *c* which has another meaning).

Deleting Lines (d)

Suppose you want to delete the line

```
This is text added in Session 2.
```

from the buffer. If you know the number of the line to be deleted, you can type that number followed by *delete* or *d.* This example deletes line 4:

```
:4d
It doesn't mean much here, but
```

Here "4" is the number of the line to be deleted and "delete" or "d" is the command to delete the line. After executing the delete command, edit prints the resulting new current line (.).

If you do not happen to know the line number, you can search for the line and then delete it using this sequence of commands:

```
:/added in Session 2./
This is text added in Session 2.
:d
It doesn't mean much here, but
```

The "`/added in Session 2./`" asks *edit* to locate and print the next line containing the indicated text. Once you are sure that you have correctly specified the line you want to delete, you can enter the delete (d) command. In this case it is not necessary to specify a line number before the "d". If no line number is given, *edit* deletes the current line (.), that is, the line found by the search operation. After the deletion, your buffer should contain:

```
This is some sample text.
And this is some more text.
Text editing is nice.
It doesn't mean much here, but
it does illustrate the editor.
```

To delete both lines 2 and 3:

```
And this is some more text.
Text editing is nice.
```

type

```
:2,3d
```

to specify the range of lines (2 thru 3) and the operation on those lines (*d* for delete).

Again, this assumes that you know the line numbers for the lines to be deleted. If you do not, you can combine the *search* and *delete* commands as follows:

```
:/And this is some/,/Text editing is nice/d
```

This tells the editor to find the first line (following the current line) that contains the characters "And this is some", then delete it and all subsequent lines until it has deleted the line containing "Text editing is nice".

Be Careful

In using the search function to locate lines to be deleted, make absolutely sure that the characters you give as the basis for the search will take *edit* to the line you want deleted. *Edit* searches for the first occurrence of the characters starting from where you last edited; that is, from the line you see printed if you type a period (.) then press Return.

A search based on too few characters may result in the wrong line being deleted (if an identical pattern appears elsewhere in the text). For this reason, it is usually safer to specify the search, then delete in a second separate step, at least until you become familiar enough with the editor that you understand how best to specify searches. For beginners, be safe and double-check each command before pressing Return to send the command on its way.

Oops! I goofed. Now what? (undo)

The *undo* (*u*) command has the ability to reverse the effects of the last (and only the last) command. To undo the previous command type `u` or `undo`. *Undo* can rescue the contents of the buffer from many an unfortunate mistake. However, its powers are not unlimited, so it is still wise to be reasonably careful about the commands you give. *Undo* affects only those commands that can change the buffer, such as delete, append, move, copy, substitute, and even undo itself. The commands *write* (*w*) and *edit* (*e*) which interact with disk files cannot be undone, nor can commands such as *print* which do not change the buffer. Most important: the only command that can be reversed by undo is the last "undo-able" command preceding the *undo*.

To illustrate, let's issue an undo command. Recall that the last buffer-changing command deleted the lines that were formerly numbered 2 and 3. Executing *undo* at this time reverses the effects of the deletion, causing those two lines to be restored to their original position in the buffer.

```
:u
2 more lines in file after undo
And this is some more text.
```

Again, as before, edit informs you when the command affects more than one line, and prints the text of the resulting new current line.

If after using *undo* you discover that the change was correct, you can *undo* the *undo* by giving another *undo* command.

More About Dot (.) and Buffer End ($)

The function assumed by the dot symbol (period) depends on its context. It can be used to:

- Exit from append mode by typing a period (by itself) followed immediately by Return
- Refer to the current line in the editor's buffer.

A period can also be combined with an equal sign to get the number of the line currently being edited (current line):

```
:.=
```

Thus, type .= to ask for the number of the current line, or use a colon instead of the equal sign (.:) to ask for the text in the current line.

In this editing session, as in the last, the dollar sign was used to to indicate the last line in the buffer for commands such as *print*, *copy*, and *move*. As a command, the dollar sign asks *edit* to print the last line in the buffer. If the dollar sign is combined with the equal sign ($=), *edit* prints the line number corresponding to the last line in the buffer.

(.) and ($) therefore represent line numbers. Whenever appropriate, these symbols can be used in place of line numbers in commands. For example:

```
:.,$d
```

instructs *edit* to delete all lines from the current line (.) through the last line in the buffer.

Moving Around in the Buffer (+ and -)

It is frequently convenient during an editing session to go back and re-read a previous line. You could specify a context search for a line you want to read if you remember some of its text, but if you simply want to see what was written a few (say, 3) lines ago, you can type:

```
-3p
```

This tells *edit* to move back to a position 3 lines before the current line (.) and print that line. You can move forward in the buffer similarly:

```
+2p
```

tells *edit* to print the line which is 2 ahead of our current position. You can use + and - in any command where *edit* accepts line numbers. Line numbers specified with "+" or "-" can be combined to print a range of lines. The command:

```
:-1,+2copy$
```

copies 4 lines: the line preceding the current line, the current line, and the two lines following the current line, placing them after the last line in the buffer ($).

Try typing a single minus (-). You will move back one line just as if you had typed, `:-1p`. Typing the command "+" works similarly. You might also try typing a few plus or minus signs in a row (such as "+++") to see edit's response. Typing a carriage return alone on a line is the equivalent of typing "+1p": it moves you one line ahead in the buffer and prints that line.

If you are at the last line in the buffer and try to move further ahead, perhaps by typing a "+" or a carriage return alone on the line, *edit* reminds you that you are at the end of the buffer:

```
At end-of-file
```

Similarly, if you try to move to a position before the first line, edit will print one of these messages:

```
Nonzero address required on this command
Negative address - first buffer line is 1
```

The number associated with a buffer line is the line's "address", in that it can be used to locate the line.

Changing Lines (c)

There may be occasions when you want to delete certain lines and insert new text in their place. This can be accomplished easily with the *change* (c) command. The change command instructs *edit* to delete specified lines then switch to text input mode in order to accept the text that will replace them. Let's assume that you want to change the first two lines in the buffer:

```
This is some sample text.
And this is some more text.
```

to read

This text was created with the HP-UX text editor.

To do so, you can type:

```
:.,2c
2 lines changed
This text was created with the HP-UX text editor.
.
:
```

The command *1,2c*, specifies that you want to change the range of lines beginning with 1 and ending with 2 by giving line numbers as with the *print* command. These lines will be deleted. After a Return enters the change command, *edit* notifies you if more than one line is being changed, then places you in text input mode. Any text typed on the following lines is inserted into the position where lines were deleted by the change command. You remain in text input mode until you exit by typing a period alone on a line. Note that the number of lines added to the buffer need not be the same as the number of lines deleted.

This is the end of the third session on text editing with HP-UX.

Session 4

This lesson covers several topics, starting with commands that affect the entire buffer, characters with special meanings, and how to issue HP-UX commands while using the editor. The next topics deal with files, discussing more about reading and writing, and explaining how to recover files lost in a crash. The final section provides leads to other sources of information and other editors that expand beyond *edit.*

Making Commands Global (g)

One disadvantage of using the commands in the manner illustrated when searching or substituting is that if you have a number of instances of a word to change, it would appear that you have to type the command repeatedly, once for each time the change needs to be made. *Edit*, however, provides a way to make commands apply to the entire contents of the buffer – the *global* (*g*) command. To print all lines containing a certain sequence of characters (say, "text") the command is:

```
:g/text/p
```

The *g* instructs edit to make a global search through the file for all lines in the buffer containing the character pattern *text.* The *p* prints the lines found.

To issue a global command, start by typing a "g" and then a search pattern identifying the lines to be affected. Then, on the same line, type the command to be executed on the identified lines. Global substitutions are frequently useful. For example, to change all instances of the word "text" to the word "material" the command would be a combination of the global search and the substitute command:

```
:g/text/s/text/material/g
```

In this example, the "g" at the beginning of the line tells *edit* to change every line in the file that contains the word or words between it and the **s** command. The "g" at the end of the line tells *edit* to make the substitution every time the word or words being changed appear in any given line. If the second "g" is absent, the substitution is made **only** on the first occurrence of the text being altered on the line identified by the pattern between the first "g" and the "s" command.

You can give a command such as:

```
:14x/text/material/g
```

to change every instance of "text" in line 14 alone. Note further that neither command will change "Text" to "material" because "Text" begins with a capital rather than a lower-case t. Edit does not automatically print the lines modified by a global command. If you want the lines to be printed, type a "p" at the end of the global command:

```
:g/text/s/text/material/gp
```

The usual qualification should be made about using the global command in combination with any other. Be sure you know what you are telling *edit* to do to the entire buffer. For example:

```
:g/ /d
72 less lines in file after global
```

deletes every line containing a blank anywhere in it. This could demolish your document, because most lines contain spaces between words, and thus would be deleted. After executing the global command, *edit* prints a warning if the command added or deleted more than one line. Fortunately, the undo command can reverse the effects of a global command. Try experimenting with the global command on a small buffer of text to see what it can do for you.

Be careful when using global substitutions. For example, in the previous illustration, the word "textual"A would be changed to "materialual" by the substitution of "material" for "text".

More about Searching and Substituting

Previous examples of using slashes to identify a character string that you want to search for or change have always specified the exact characters. There is a less tedious way to repeat the same string of characters. To change "noun" to "nouns" you can type either

```
:/noun/s/noun/nouns/
```

as before, or use a somewhat abbreviated command:

```
"/noun/s//nouns/
```

In this example, the characters to be changed are not specified (there are no characters, not even a space, between the two slash marks that indicate what is to be changed). This lack of characters between the slashes is taken by the editor to mean "use the characters we last searched for as the characters to be changed".

Similarly, the last context search can be repeated by typing a pair of slashes with nothing between them:

```
"/does/
It doesn't mean much here, but
://
it does illustrate the editor
```

Because no characters are specified for the second search, the editor scans the buffer for the next occurrence of the characters "does".

Edit normally searches forward through the buffer, wrapping around from the end of the buffer to the beginning, until the specified character string is found. If you want to search in the reverse direction, use question marks (?) instead of slashes to surround the character string.

It is also possible to repeat the last substitution without having to retype the entire command. An ampersand (&) used as a command repeats the most recent substitute command, using the same search and replacement patterns. After altering the current line by typing

```
:s/noun/nouns/
```

you could use the command

```
:/nouns/&
```

or simply

```
://&
```

to make the same change on the next line in the buffer containing the characters "nouns".

Special Characters

Two characters have special meanings when used in specifying searches: the dollar sign ($), and circumflex (^). ($) is taken by the editor to mean "end of the line" and is used to identify strings which occur at the end of a line.

```
:g/ing&/s//ed/p
```

tells the editor to search for all lines ending in "ing" (and nothing else, not even a blank space) to change each final "ing" to "ed" and print the changed lines.

The circumflex (^) indicates the beginning of a line. Thus,

```
:s/^/1. /
```

instructs the editor to insert "1." and a space at the beginning of the current line.

These characters, ($) and (^), have special meanings only in the context of searching. At other times, they are ordinary characters. If you ever need to search for a character that has a special meaning, you must indicate that the character is to temporarily lose its special significance by typing another special character, the backslash (\), before it.

```
:s/\$/dollar/
```

looks for the character "$" in the current line and replaces it by the word "dollar". Were it not for the backslash, the "$" would have represented "the end of the line" in your search, rather than the character "$". The backslash retains its special significance unless it is preceded by another backslash.

Issuing HP-UX Commands from the Editor

After creating several files with the editor, you may want to delete files no longer useful to you or ask for a list of your files. Removing and listing files are not editor functions, so they require use of HP-UX system commands (also referred to as "shell" commands, because the HP-UX program that processes HP-UX commands is called a "shell"). You do not need to quit the editor to execute an HP-UX command as long as you indicate that it is to be sent to the shell for execution. To use the HP-UX command *rm* to remove the file named *junk*, type:

```
:!rm junk
!
:
```

The exclamation point (!) indicates that the rest of the line is to be processed as an HP-UX command. If the buffer contents have not been written since the last change, a warning is printed before the command is executed. The editor replies with an exclamation point when the command is completed. The *Getting Started with HP-UX* manual describes useful features of the system, and is helpful background when you need to access HP-UX from *edit*.

Filenames and File Manipulation

Throughout each editing session, *edit* keeps track of the name of the file being edited as the current filename (the current filename is the name given when you entered the editor). The current filename changes whenever the edit (e) command is used to specify a new file. Once *edit* has recorded a current filename, it inserts that name into any command where a filename has been omitted. If a write command does not specify a file, edit, as you have seen, supplies the current filename. You can have the editor write all or part of its buffer contents to a different file by including the new file name in the *write* command:

```
:w chapter3
"chapter3" 283 lines, 8698 characters
:
```

The current filename remembered by the editor does not change as a result of the *write* command unless it is the first filename given in the editing session. Thus, using the previous example, the next *write* command that does not specify a file name will write onto the current file, not onto the file *chapter3*.

The File (f) Command

To ask for the current filename, type *file* (or *f*). In response, the editor provides updated information about the buffer, including the filename, your current position, and the number of lines in the buffer:

```
:f
"text" [Modified] line 3 of 4--75%--
```

If the contents of the buffer have changed since the last time the file was written, the editor will tell you that the file has been "Modified". After you save the changes by writing to a disk file, the buffer is no longer considered modified:

```
:w
"text"4 lines, 88 characters
:f
"text"line 3 of 4--75%--
```

Reading Additional Files (r)

The read (r) command enables you to add the contents of a file to the buffer without destroying the text already there. To use it, specify the line after which the new text is to be placed, the command *r*, then the name of the file.

```
:$r bibliography
"bibliography" 18 lines, 473 characters
```

This command reads in the file *bibliography* and adds it to the buffer after the last line. The current filename is not changed by the *read* command unless it is the first filename given in the editing session.

Writing Parts of the Buffer

The *write* (*w*) command can write all or part of the buffer to any file you specify. You are already familiar with writing the entire contents of the buffer to a disk file. To write only part of the buffer onto a file, indicate the beginning and ending lines before the write command. For example:

```
:45,$w ending
```

Here all lines from 45 through the end of the buffer are written to the file named *ending*. The lines remain in the buffer as part of the document you are editing, so you can continue to edit the entire buffer.

Recovering Files

Under most circumstances, *edit*'s crash recovery mechanism is able to save work to within a few lines of changes after a crash or if your terminal is accidentally diskonnected. If you lose the contents of an editing buffer in a system crash, you will normally receive mail when you login, listing the name of the recovered file. To recover the file, enter the editor and type the command *recover* (*rec*), followed by the name of the lost file.

```
:recover chap6
```

Recover is sometimes unable to save the entire buffer successfully, so always check the contents of the saved buffer carefully before writing it back onto the original file.

Other Recovery Techniques

If something goes wrong while you are using the editor, it may be possible to save your work by using the command *preserve* (*pre*), which saves the buffer as if the system had crashed. If you are writing a file and receive the message "Quota exceeded", you have tried to use more disk storage than is allotted to your account. Proceed with caution because it is likely that only a part of the editor's buffer is now present in the file you tried to write. In this case, you should use the shell escape from the editor (!) to remove some files you don't need and try to write the file again. If this is not possible and you cannot find someone to help you, enter the command

```
:preserve
```

then seek help. Do not simply leave the editor. If you do, the buffer will be released (and possibly destroyed), and you may not be able to save your file. After a *preserve*, you can use the *recover* command once the problem has been corrected.

If you make an unwanted change to the buffer and issue a *write* command before diskovering your mistake, the modified version will replace any previous version of the file. Should you ever lose a good version of a document in this way, do not panic and leave the editor. As long as you stay in the editor, the contents of the buffer remain accessible. Depending on the nature of the problem, it may be possible to restore the buffer to a more complete state with the *undo* command. After fixing the damaged buffer, you can again write the file to disk.

Further Reading and Information

Edit is an editor designed for beginning and casual users. It is actually a version of a more powerful editor called *ex*. These lessons are intended to introduce you to the editor and its most commonly used commands. We have not covered all of the editor's commands, just a selection of commands which should be sufficient to accomplish most of your editing tasks. You can find out more about the editor in the *ex* tutorial, which is applicable to both *ex* and *edit*. One way to become familiar with *ex* is to begin by reading the description of commands that you already know.

Using ex

As you become more experienced with using the editor, you may still find that *edit* continues to meet your needs. However, should you become interested in using *ex*, it is easy to switch. To begin an editing session with *ex*, use *ex* in your command instead of *edit*.

Edit commands work the same way in *ex*, but the editing environment is somewhat different. You should be aware of a few differences that exist between the two versions of the editor. In *edit*, only the characters ^, **$**, and \ have special meanings in searching the buffer or indicating characters to be changed by a *substitute* command. Several additional characters have special meanings in *ex*, as described in the *ex* tutorial. Another feature of the *edit* environment prevents users from accidentally entering two alternative modes of editing, **open** and **visual**, in which the editor behaves quite differently than in normal command mode. If you are using *ex* and the editor behaves strangely, you may have accidently entered *open* mode by typing *o*. Type the ESC key and then a "Q" to get out of open or visual mode and back into the regular editor command mode. *The Vi/Ex Editor* tutorial earlier in this volume provides a full discussion of visual mode.

Index

a

b

c

d

e

f

g

i

j

l

m

n

o

p

r

s

t

u

v

w

Table of Contents

AWK: A Programming Language for Manipulating Data

Introduction

Awk is a useful tool for manipulating data and text. Unlike the HP-UX commands that do similar work, *awk* comprises its own programming language. This lets you process input in various ways, such as:

- Generate reports on the contents of files
- Transform the text or data within files
- Manipulate columnar data
- Search files for specific patterns

With *awk*'s ability to search files and generate reports, you can treat some of your ordinary files as databases. The terminology used in *awk* — "records" and "fields" — reinforces this idea.

The *awk* programming language includes such constructs as **for**, **while**, and **if-else**, as well as a set of built-in functions and variables. The language resembles the C programming language. If you are familiar with C, you should be able to master *awk* almost immediately. If you don't know C, you should still find *awk* easy to learn and use.

Awk is named for its designers: Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan, from Bell Laboratories. For a detailed discussion written by these people, read "Awk—A Pattern Scanning and Processing Language," published by Bell Labs in 1978 and available in many technical libraries.

This article is for the user who is familiar with HP-UX and who has used a programming language. There are examples throughout the article; you should try them as you encounter them. You should take time now to create a small input file, using any of the HP-UX editors, or by typing the following command line:

```
$ cat >hello.awk
```

The *cat* command followed by > allows you to type text directly into the file *hello.awk* from the keyboard. The filename is arbitrary; the *.awk* suffix is just a reminder for you and is optional.

Now type in

```
Hello, world!
Howdy, partner!
```

End the file by typing **CONTROL** and [D] together (the end-of-file character) and then pressing [Return]. (**CONTROL** may be marked [CTRL] or [CTRL] on your keyboard; [Return] is marked [Enter] on some keyboards with HP-UX overlays.) The shell prompt should reappear; your example file is ready to use.

The Command Line

You can program *awk* entirely on the same line with the prompt. This is often done in practice. The format is:

```
$ awk 'awk_program' input_filename
```

The dollar sign at the begining of this command line represents the shell prompt in this tutorial. Your shell prompt may be different.

Command-line *awk* programs must be surrounded by single or double quotes, so that the shell will see the whole program as a single argument to the command. Single quotes prevent the shell from interpreting any special characters you may have included in the *awk* program. All examples in this tutorial use single quotes. For more information on shell quoting rules, read "UNIX Programming" in Volume 2 of *HP-UX Concepts and Tutorials.*

If your *awk* program exceeds one line, you can type a backslash (\), then press the [Return] key, and continue typing the program. For example:

```
$ awk 'awk_pro\ [Return]
> gram' in\ [Return]
> put_filename [Return]
output of program
$
```

The > is an auxiliary prompt (which may be different on different systems or shells) that tells you you're still typing a single logical command line.

This maneuver is called *escaping the newline character.* You can use it when invoking any command from the shell.

For some applications you may want to write *awk* programs that are many lines long. It makes sense to store such long programs, and any programs that you often use, in separate files. Note that no compilation step is necessary. The file doesn't have to be executable, just readable.

To invoke *awk* using a program stored in a separate file, use the **–f** option:

```
$ awk -f awk_program_name input_filename
```

You can give an *awk* program input from your keyboard (standard input) by typing a dash ("–") instead of an input filename. Keyboard input is terminated by typing **CONTROL-**[D]. Your command line would look like this:

```
$ awk 'awk_program' -
```

or

```
$ awk -f awk_program_name -
```

An example of this procedure is shown in the Annotated Example, "Doing Calculations."

Structure of Awk Programs

Awk programs are built of one or more *statements* that have the general form:

```
pattern {action}
```

For every line in the input that matches the pattern, the specified action is executed. The action part is always enclosed in braces.

You can specify multiple actions within the action part by separating them with semicolons or newline characters (typing Return creates a newline character).

Awk processes input one line at a time. For *each* line of input, *awk* scans *all* the patterns in the program. Whenever it finds a pattern that matches the line of input in question, it executes the associated action.

An *awk* statement may consist of the pattern or the action or both. A pattern without an action prints out each input line that matches the pattern (this is the default action); an action without a pattern executes the action on every line of input (the default pattern matches anything).

Predefined Variables

The input is made up of a series of *records*. The default record separator is a newline character; by default, each input line is a record.

Records are divided into *fields*; the default field separator is white space (tabs or blanks). So the input

```
Hello, world!
```

consists of one record (one line) and two fields (the strings "Hello," and "world!", which are separated by a blank).

The output is also made up of fields and records. The default output field separator is a blank and the default output record separator is the newline character.

The variables *FS* and *RS* contain the current input field and record *separators*; the output separators are in *OFS* and *ORS*. You can change any of them at any time by simply assigning them any single character value.

On the command line, you can use the argument **−Fc**, which sets *FS* to the character value *c*. Use assignment statements (such as RS = "@") to specify new values for any of the other predefined variables or as an alternate way to change *FS*. (Be sure to put double quotes around new separators to ensure that they are interpreted correctly.)

Each field is designated by a *field variable*. In the first record of *hello.awk*, the string "Hello," is stored in the field variable *$1* and "world!" is stored in the field variable *$2*. In general, field *n* of the current record is stored in the variable *$n*. The whole current record is stored in *$0*.

A predefined variable called *NF* contains the number of fields in the current record. The number of the record currently being processed is stored in *NR*; you can find out how many records are in the input by printing *NR* at the end of your program.

Output

The simplest type of *awk* program prints out each line in an input file that matches a specified string. Try this command:

```
$ awk '/Hello/' hello.awk
```

There is no action supplied here, so each record (line) that contains "Hello" somewhere within it is printed. Note that the string is surrounded by slashes, and that the whole *awk* program is surrounded by single quotes. You must always use these slashes around patterns which consist of strings that are regular expressions (described in the section of this article entitled "Regular Expressions and Special Characters"), and you should use single quotes around a command-line program so the shell will see it as one argument and not attempt to interpret any special characters that may be lurking within the program.

The output for the above command is the matching record:

```
Hello, world!
```

The following program contains an action, but it does the same thing as the above actionless program:

```
$ awk '/Hello/ {print $0}' hello.awk
```

This is an example of the **print** action. Since *$0* refers to the entire record, this program prints every record containing "Hello" on the standard output. To print out the second and first fields, in that order, of each record containing "Hello", type:

```
$ awk '/Hello/ {print $2, $1}' hello.awk
```

and you'll get:

```
world! Hello,
```

The comma between the field arguments tells *awk* to put an output field separator (a space by default) between the output fields. Without the comma, the fields would be concatenated (run together).

Redirecting Output to Files

You can send the output of the **print** action into files by using > or >>. The program

```
$ awk '/Hello/ {print $1 >"file1"; print $2 >"file2"}' hello.awk
```

writes the first field, "Hello,", into *file1* and the second, "world!", into *file2* (creating the files if necessary). You must put double quotes around the file names. The program:

```
$ awk '/Hello/ {print $1 >>"file1"}' hello.awk
```

appends the first field to *file1* rather than overwriting it, so now *file1* contains:

```
Hello,
Hello,
```

The file name to which you divert your output may also be a variable or a field. The action:

```
$ awk '/Hello/ {print NF > $2}' hello.awk
```

uses *$2* for the filename. You should take care in cases like this one that the value assigned to the variable is a valid file name. If it is not, you will get an error message and the program will abort.

Formatting Output

You can format your output with the **printf** statement. The *awk* **printf** statement is identical to the **printf** library routine used in the C programming language. The statement's structure is

printf *format, expr, expr, ...*

The format for the list of expressions is specified in the *format* argument. **Printf** prints the expressions in the specified format. For example,

```
$ awk '{printf "%7s  %10.3f\n", $1, NF}' hello.awk
```

prints *$1* (the first field) as a seven-character string, and *NF* as a floating-point number in a ten-digit field width with three digits after the decimal point. Try this and get:

```
Hello,       2.000
```

The newline character is **\n**, which appears at the end of the *format*. You must specify all spaces, separators, and newlines that you want in the output. Note that you don't have to specify a newline when using **print**, because **print** automatically appends the output record separator (by default, a newline) to its output string.

For a full discussion of **printf**, look in McGilton and Morgan's *Introducing the UNIX™ System*, Kernighan and Ritchie's *The C Programming Language*, or the article "Using the C Library Routines" in *HP-UX Concepts and Tutorials*. (These are listed in a reference section at the end of this tutorial.)

Details of Awk Programming

The full structure of *awk* programs includes optional statements labeled by the special patterns *BEGIN* and *END*:

```
BEGIN    { action }
  .          .
  .          .
  .          .
pattern  { action }
  .          .
  .          .
  .          .
END      { action }
```

The action in the *BEGIN* statement is executed once before any of the input has been read (hence before any patterns are evaluated). The action in the *END* statement is executed once after all the input has been read. These special statements give you opportunities to set parameters before the program begins or to process or tabulate data after *awk* has seen all of the input. For example,

```
BEGIN {OFS = @}
END   {print NR}
```

changes the output field separator to "@" before any input is read, and prints out how many records are found in the input after all of it has been read.

Designing Patterns

You have many options for writing *awk* patterns, including:

- Regular expressions
- Relational expressions
- Combinations of expressions
- Boolean expressions
- Ranges of patterns

You have a complete set of operators and special characters with which to build patterns.

Regular Expressions and Special Characters

Patterns can be made from regular expressions. Regular expressions are always enclosed in slashes. A simple pattern is a string enclosed in slashes:

```
/world/
```

If entered as a program (`$ awk '/world/' hello.awk`) this expression would print out all lines in an input containing occurrences of "world", both as a field alone (a complete word) and as part of a field, such as "worldly" or "world!"

Between the slashes that delimit regular expressions, you can use most of the standard special characters (or metacharacters) that are recognized by the *ed* editor and by the shell. The available special characters for use between slashes in regular expressions are:

|	perform a logical OR of the regular expressions on either side of the vertical bar.
+	match if there are one or more occurrences of the preceding regular expression.
?	match if there are zero or one occurrence(s) of the preceding regular expression.
[ ]	match any of the characters inside the brackets.
[*x*-*x*]	match any character in the lexical range bounded by the characters on either side of the dash. The range is enclosed by the brackets.
^	match only if the matching regular expression is found at the beginning of the line.
$	match only if the matching regular expression is found at the end of the line.

*	match a succession of zero or more of the preceding single-character regular expression.
.	match any one character in the position of the period.
\	turn off the special meaning of the next character, so the character can represent itself (a maneuver known as escaping the character).
()	group the evaluation of regular expressions

For example,

```
$ awk '/^main/' c_program.c
```

matches records beginning with "main", and:

```
$ awk '/Albuquerque|Santa Fe/' article_about_NM
```

matches records containing a reference to either Albuquerque or Santa Fe.

To turn off a special character's special meaning, precede it with a backslash in the expression. For example,

```
/\/.*\//
```

matches any string of one or more characters that is enclosed in slashes.

You can abbreviate a sequence of characters in a string. This is called *character class abbreviation.* For example,

```
$ awk '/[Hh]ello/' hello.awk
```

matches:

```
Hello, world!
```

but it would also have matched a record containing:

```
Well, world, hello!
```

The sequence `[a|-zA|-ZO|-9]` would match all letters, both upper and lower case, and all digits. To use such ranges you need to understand how your character set is arranged. McGilton and Morgan explain this in their book.

In patterns, you can specify that a field or variable (an expression) matches a regular expression using the tilde character "~" to mean "match" and "!~" to mean "don't match." For example, the pattern:

```
$ awk '$1 ~ /[Hh]ello/' hello.awk
```

matches all records that contain either "Hello" or "hello" in the first field. This pattern also matches records containing "Othello" in the first field. To *reject* records with "Hello", use

```
$ awk '$1 !~ /[Hh]ello/' hello.awk
```

Relational Expressions

In *awk* patterns, you can use the relational operators <, <=, ==, !=, >=, and > between expressions. For example, the pattern

```
$ awk '$2 >= $1 + 100' filename
```

selects lines in which the second field is numerically at least 100 greater than the first field.

Relational operations are always *numeric* comparisons (as in the above example) unless *both* operands are strings; in that case a *string* comparison is made. Fields are treated as strings unless there is information to the contrary, so

```
$ awk '$1 > $2' filename
```

automatically performs a string comparison on the first two fields, matching if *$1* has a larger character value (in ASCII) than *$2*.

Note that the regular expressions in the last two examples of the Regular Expressions and Special Characters section match the string "Hello," in the *hello.awk* file as well as "Hello" or "hello", or even "helloes". You can eliminate these various matchings by using a string instead of a regular expression:

```
$ awk '$1 == "Hello," || "hello," ' hello.awk
```

matches only if *$1* is either the string "Hello," or "hello," and nothing else. This generates the same output as the program

```
$ awk ' $1 ~ /^[Hh]ello,$/ ' hello.awk
```

which specifies that *$1* starts with "H" or "h" and ends with a comma. In this case it's shorter to use the regular expression.

If you use regular expressions in your patterns, you can match many strings. But if you use strings in your patterns, you can match only those exact strings in the input. Both tactics are valuable in different situations.

Combinations of Patterns

You can combine several patterns into one using the Boolean operators | (or), && (and), == (equal to), and != (not equal to). For example, the pattern

```
$ awk '$1 >= "H" && $1 < "I" && NF == 2 && $2 != "world!"' hello.awk
```

matches records that begin with "H" and have two fields but do not have "world!" as the second field. The record "Hello, world!" won't match, but the record "Howdy, partner!" (or "Houston, Texas" for that matter) will match.

Awk always evaluates the operands of && and | from left to right. The evaluation stops as soon as the expression is found to be true or false. You can use parentheses freely to force the order of evaluation or to increase legibility.

Pattern Ranges

The pattern you specify in a pattern-action statement can consist of two patterns separated by a comma. When you specify the pattern in this way, the action is executed on each record from an occurrence of the first pattern through the next occurrence of the second pattern. For example,

```
$ awk '/Hello/,/partner/' hello.awk
```

prints all records from the first one matching "Hello" through the next one matching "partner". The statement

```
$ awk 'NR == 10, NR == 30 {print $0}' filename
```

prints records 10 through 30 of some file (try it on one of your files). If you use the above program on a 20-line file, *awk* will print lines 10 through 20 and stop without generating an error.

Designing Actions

Actions consist of one or more statements. A statement can include:

- Arithmetic expressions
- Assignment statements
- Output statements
- Built-in function calls
- Flow-of-control statements

Variables

In *awk* programs, you do not need to write declaration statements for variables. The variables take on numeric (floating point) or string values automatically, according to context. For example,

`x = 1`	gives *x* a numeric value;
`x = "HP-UX"`	gives *x* a string value;
`x = "3" + "4"`	assigns 7 to *x* as if the equation were *x = 3 + 4* (because context demands numeric values in this case).

Variables are automatically initialized to the null string (numerical value = 0), so you don't need to initialize variables in a *BEGIN* statement. For example, the sums of the first two fields of all records can be computed by a two-line program:

```
      { s1 = s1 + $1; s2 = s2 + $2 }
END   { print s1, s2 }
```

which you can enter and then try by typing

```
$ awk -f two_line_program file_with_numbers
```

Awk does all of its arithmetic internally and in floating point. The available operators are:

+, −, *, and /	addition, subtraction, multiplication, and division
%	the mod operator (for example, the pattern *NF % 2 == 0* prints all lines in the input that have an even number of fields)
++ and −−	the increment and decrement operators (like those in C language)

+=, −=, *=, /=, and %= the C language assignment operators (for example, *x += 1* is the same as *x = x + 1*).

Any of these operators may be used in an expression.

Field Variables

You can treat the field variables (*$1*, *$2*, etc.) just as any other variable. You can replace fields with other numbers, assign results to a field, or use fields in expressions. For example,

```
$ awk '{ $1 = $2 + $3; print $0 }' filename
```

accumulates fields 2 and 3 into field 1 and prints out the record with a new field 1. If you use *hello.awk* for *filename*, *awk* will convert the strings to numbers in response to the context and *$1* will turn out to be zero. *$3* is a null string which equals zero.

Fields can be referred to by numerical expressions, such as *$(i)*, *$(n+1)*, or *$(NF*4 + 3/(NR−5))*. (If the expression comes out non-integer, *awk* truncates the decimal portion and uses the remaining integer portion as the result.) For example, to refer to the last field when you're unsure how many fields are in the record, use *$NF*.

Whether a field variable is considered numeric or string depends on context. The matter is not a concern to most *awk* users. You may run into ambiguous cases such as

```
if ($1 == $2)
```

in which *awk* has no criteria for deciding whether to compare strings or numbers. Just as in the relational expressions discussed earlier, *awk* solves the ambiguity by treating fields as strings in such cases.

Arrays

Awk also defines and initializes arrays automatically. To create an array, simply mention it when you need it; *awk* creates the array for you then and there. The subscripts can have numeric values or string values, such as *x*["Hello,"]. The program

```
/Hello/   {x["Hello"]++}
/world/   {x["world"]++}
END       {print x["Hello"], x["world"]}
```

counts the occurrences of "Hello" and "world" in the input, stores the counts in elements of the array, and prints the final results. Enter this progam in a file, and try it using the −f option.

Built-in Functions

You can use a number of built-in functions in your *awk* programs. These include both string and arithmetic operations.

length(*x*) returns the length of the argument. For example,

```
$ awk '{print length($1), $0}' hello.awk
```

prints the length of the first field then prints out the entire record.

sqrt(*x*) returns the square root of the argument.

log(*x*) returns the base *e* logarithm of the argument.

exp(*x*) returns the exponential of the argument.

int(*x*) returns the integer part of the argument.

The arguments of functions can be any expression. For all of the above functions, the name of the function alone, with no argument, will cause the function to be performed on the entire record.

substr(*s,m,n*) returns the substring of *s* that begins at postion *m* and is at most *n* characters long. For example,

```
$ awk '/Hello/ {print substr($2,3,5)}' hello.awk
```

will produce

```
rld!
```

which is the substring of *$2* — "world!" — starting with the third letter — "r" — and is no more than five characters long (the length of this substring happens to be four).

split(*s,array,sep*) splits the string *s* into *array*[1],...,*array*[n]. (*s* can be a variable.) The number of elements found is returned as *n*. If you don't provide a field separator in the *sep* argument, the current value of *FS* is used by default.

index(*s1,s2*) returns the position in which the string *s2* occurs in the string *s1*. If *s2* is not a subset of *s1*, **index** returns a 0. For example,

```
$ awk '/world/ {print index($2,"r")}' hello.awk
```

prints out *3*, because "r" is the third character in *$2* ("world!").

sprintf(*f,e1,e2...*) places the values of *e1*, *e2*, and so on into the formatted fields specified by *f*. The argument *f* is the format string, which is like the **printf** format string. For example,

```
$ awk '{x = sprintf("%8s %10s", $1, $2);\
> print x}' hello.awk
```

sets *x* to the string produced by formatting strings *$1* and *$2* and prints the result. For a complete discussion of output formatting, look in "Using the C Library Routines" in Volume 2 of *HP-UX Concepts and Tutorials* or Kernighan and Ritchie's *The C Programming Language.*

The other built-in functions that you have already seen are:

print	introduced in the Output section of this article
printf	introduced in the Formatting Output section

Flow-of-Control Statements

You can use many of the same flow-of-control statements available in C (see *The C Programming Language*). *Awk* provides **if-else**, **while**, and **for** statements, and statement grouping with braces just as in C.

if(*cond*) *stmt* the condition in parentheses is evaluated; if it's true, the statement following the **if** is executed. Multiple statements are enclosed in braces and separated by semicolons or newlines. The braces are optional if there is only one statement.

else *stmt* The optional **else** statement is executed if the **if** condition is false. Multiple statements are enclosed in braces and separated by semicolons or newlines. The braces are optional if there is only one statement. For example,

```
$ awk '{if($1 != "Hello,") print $0;\
> else print "Argh!"}' hello.awk
```

prints lines that do not start with "Hello," and prints a complaint when it encounters a line that does. (Note the use of the backslash to fit this long program onto a single command line.)

while(*cond*) *stmt* The condition in parentheses is evaluated; as long as it is true, the statements in braces are executed. Multiple statements are enclosed in braces and separated by semicolons or newlines. The braces are optional if there is only one statement. The **while** condition is tested before each pass. Try this example:

```
$ awk '{while(i<=2) {print $(i); i++ };\
> i=0}' hello.awk
```

for(*cond*) *stmt* While a variable changes from an initial to a final value, the statement(s) in the braces are executed. Multiple statements are enclosed in braces and separated by semicolons or newlines. The braces are optional if there is only one statement. Here is the format of the condition:

for (initialize; final; increment){ ... }

Also, you can use:

for (i in *array*) statement

This construction executes *statement* for each element in the specified array. The elements are not necessarily accessed in order. Changing *i* or accessing any new elements during the statement will introduce side effects.

The conditional expression used in the **if**, **while**, and **for** statements can contain any of the standard relational operators (<, <=, >, >=) as well as the match operators ~ and !~ and the logical operators |, &&, ==, and !=. Parentheses for grouping are allowed (and encouraged).

Here are the other flow-of-control statements:

break	exits the current **while** or **for** construct.
continue	immediately starts the next iteration of the current loop.
next	causes *awk* to skip immediately to the next record and begin scanning the patterns from the beginning of the program.
exit	causes the program to behave as if the end of the input had occurred (thus **exit** causes execution of the *END* statement if there is one)

Commenting

Comments in *awk* programs begin with the # character and end with the end of the line:

```
/string/ {print $2, $5} #Print fields 2 and 5 if string matches
```

Error Messages

Diagnostic output for *awk* is sparse and cryptic. Most *awk* errors that stop the program are syntax errors. Typical error statements are:

```
syntax error near line 1
illegal statement near line 1
```

Syntax errors often produce an additional message:

```
bailing out near line 1
```

meaning that the program gave up and returned control to the shell.

"Near" means that the line specified in the error message may not be the line that contains the actual error.

The message

```
funny variable xxxxxxxx
```

is *awk*'s response to a variable it can't deal with, such as a the negative field variable *$(−1)*.

Some *awk* messages are more specific:

```
newline in character class near line 1
```

states the problem clearly.

Except when redirecting output, if you refer to a file that doesn't exist or can't be opened, you'll get the shell message:

```
awk:  can't open file
```

Notes on the Design

Awk improves on *grep*, *egrep*, *fgrep*, *sed*, and *ed* by offering numeric processing, logical relations, and variables. *Awk* programs do not require compilation, as C programs do, and you do not need to know the C programming language to use *awk* (though it sometimes helps). *Awk* is one of the few tools on HP-UX that let you conveniently access fields within a line (*cut* is another such tool).

The designers of *awk* tried to integrate strings and numbers, treating all quantities as both, and postponing a choice until the last minute. This is why you can generally ignore the difference between a string and a number as you write a program.

Most *awk* users extract or manipulate information from the inputs. These usages are sometimes referred to as report generation and data transformation.

Notes on Awk Implementation

Aho, Weinberger, and Kernighan wrote *awk* using tools available on HP-UX, including *yacc* and *lex*. The elements that recognize regular expressions are deterministic finite automata, constructed directly from the expressions. When you invoke *awk*, your program is translated into a parse tree by the parser that was generated by *yacc* and *lex*. A simple interpreter executes the parse tree.

Awk is not fast. Breaking input into fields and delaying the evaluation of variable types are inherent bottlenecks. Further, there is no *awk* compiler, so you cannot use faster compiled versions of oft-used programs. The *awk command* is a machine that translates (parses) and interprets a program written in the *awk language* each time the program is run.

Annotated Examples

Generating Reports

One of the practical applications of *awk* is to put text into a different form or to alter its format for a particular requirement. This example shows how text can be selectively extracted and manipulated with *awk*.

The input file is a list of universities from the Big 8, Big 10, and Pac 10 athletic conferences. (If you wish to test this example, you must type in part or all of the file, or one like it.) The file lists the universities' names (one or two fields), the states in which they are located, and the seating capacities of their stadiums. The name of this file is *schools*. You can print the file with:

```
$ awk '{print $0}' schools
Arizona          AZ       Pac 10       52000
Arizona State    AZ       Pac 10       70021
Southern Cal     CA       Pac 10       92516
Stanford         CA       Pac 10       84892
UCLA             CA       Pac 10       92516
Washington       WA       Pac 10       59800
Washington State WA       Pac 10       40000
Oregon           OR       Pac 10       41009
Oregon State     OR       Pac 10       40593
California       CA       Pac 10       76780
Michigan         MI       Big 10      101701
Ohio State       OH       Big 10       85290
Indiana          IN       Big 10       52354
Iowa             IA       Big 10       66000
Illinois         IL       Big 10       70906
Michigan State   MI       Big 10       76000
Minnesota        MN       Big 10       62212
Northwestern     IL       Big 10       49256
Purdue           IN       Big 10       69250
Wisconsin        WI       Big 10       77280
Oklahoma         OK       Big 8        75008
Oklahoma State   OK       Big 8        50817
Missouri         MO       Big 8        62000
Iowa State       IA       Big 8        50000
Colorado         CO       Big 8        51805
Nebraska         NE       Big 8        73650
Kansas State     KS       Big 8        42000
Kansas           KS       Big 8        51500
$
```

Suppose you became interested in how many of these 28 schools (print out *NR* to verify that) were located in a particular state. Because each record contains the two-letter abbreviation of the school's state, the command:

```
$ awk '/CA/ {print $0}' schools
```

results in:

```
Southern Cal      CA      Pac 10      92516
Stanford          CA      Pac 10      84892
UCLA              CA      Pac 10      92516
California        CA      Pac 10      76780
```

Similarly, you can print all the records from a particular conference, or all the schools with "State" in their names:

```
$ awk '/State/ {print $0}' schools
Arizona State     AZ      Pac 10      70021
Washington State  WA      Pac 10      40000
Oregon State      OR      Pac 10      40593
Ohio State        OH      Big 10      85290
Michigan State    MI      Big 10      76000
Oklahoma State    OK      Big 8       50817
Iowa State        IA      Big 8       50000
Kansas State      KS      Big 8       42000
$
```

In the last two examples, pattern matching was done using regular expressions. This could have backfired if the file had included records on, say, the Central YMCA Community College of Chicago (first example) or of a college on Staten Island (second example). When designing patterns you should be aware of such potential pitfalls and think of ways around them.

Printing out all these records (lines) may not be all that you want to do. Suppose you're working for a professional soccer league that's looking for stadiums. You've been assigned to find stadiums with seating capacities greater than 60,000. The *schools* file is a limited resource, but it's a fair place to start. To find out which stadiums are big enough for your needs, use an action containing an **if** statement. Because you want to test every record, a pattern is not necessary. Type:

```
$ awk '{if ($(NF) > 60000) print $0}' schools
```

and get:

```
Arizona State     AZ      Pac 10      70021
Southern Cal      CA      Pac 10      92516
Stanford          CA      Pac 10      84892
UCLA              CA      Pac 10      92516
California        CA      Pac 10      76780
Michigan          MI      Big 10     101701
Ohio State        OH      Big 10      85290
Iowa              IA      Big 10      66000
Illinois          IL      Big 10      70906
Michigan State    MI      Big 10      76000
Minnesota         MN      Big 10      62212
Purdue            IN      Big 10      69250
Wisconsin         WI      Big 10      77280
Oklahoma          OK      Big 8       75008
Missouri          MO      Big 8       62000
Nebraska          NE      Big 8       73650
```

Remember, *NF* is the predefined variable that means the number of fields in the current variable. You could do this job with just a pattern:

```
$ awk ' $NF > 60000 ' schools
```

gets the same result (and it's shorter). If you want to save this information, type

```
$ awk '{if ($NF > 60000) print $0 >>"big_stadiums"}' schools
```

The >> operator appends the output to *big_stadiums*; you should use this operator as you search other files for similar information. Note the use of *$NF* for the stadium-capacity field. This is used because the number of fields per record varies, but the stadium capacity is always in the last field.

Assume you want to find the average size of a group of stadiums. You need to scan all the records to add up the stadium sizes, then divide the total by the number of records to get the average. The final calculation takes place in an *END* statement after you have processed all of the input. Due to the size of this *awk* program, place it in a separate file:

```
$cat >avg_capacity
{ x += $(NF) }  # accumulate capacities in x
END { x /= NR; print "Average Capacity = ", x}
<control-d>
$
```

The **<control-d>** line represents typing **CONTROL-D**.

No quotes are needed around the program to protect it from shell interpretation, because you are not typing it on the command line. Also, you do not need to initialize x; *awk* sets x to the null string when it is created. The arithmetic is automatically done in floating point. To run the program, use the **–f** option:

```
$ awk -f avg_capacity schools
Average capacity = 64898.4
$
```

A longer program finds the average stadium capacity of each conference:

```
/Pac/      {xp += $(NF); ip++ }     #accum caps in xp; incr. count
/Big 10/   {x10 += $(NF); i10++}    #accum caps in x10; incr. count
/Big 8/    {x8 += $(NF); i8++}      #accum caps in x8; incr. count
END        {print "Avg. cap. Pac 10 = ", (xp/ip)
            print "Avg. cap. Big 10 = ", (x10/i10)
            print "Avg. cap. Big 8 = ", (x8/i8)
           }
```

The calculations and print statements for each conference cannot be on the same line with the accumulation statements because *awk* runs all the pattern-action pairs on each input record as it arrives. *END* is executed after all the input comes in. You don't know all the data until the *END* statement.

Notice that no semicolons are used in the multi-line *END* statement. *Awk* treats the newline as another expression separator.

The output of this program is

```
Avg. cap. Pac 10 = 65012.7
Avg. cap. Big 10 = 71024.9
Avg. cap. Big 8  = 57097.5
```

The information in the file *schools* forms a small database. *Awk* is useful for retrieving and manipulating information from such databases. Other applications include updating or reformatting input files.

One last job. Who has the biggest stadium?

```
$cat >findbiggest
{ x = $(NF); if (x>y) {y=x; bigrecord = $0}}
END {print bigrecord}
<control-d>
$

$ awk -f findbiggest schools
Michigan          MI      Big 10       101701
$
```

Doing Calculations

The following program finds the mean and the square root of the sum of the squares (root-mean-square or rms) of a list of input numbers. The work is performed on every record, so there are no patterns in this program other than *END*. This example is taken from "A Walk Through Awk," a paper by Leon S. Levy of Bell Laboratories.

```
{sum_of_squares += $1 * $1}  #accum sum of squares
{sum += $1}                  #accum nos for mean calc
END   {mean = sum / NR       #calc mean
       print "mean = ", mean
       rms = sqrt(sum_of_squares/NR)  #calc rms
       print "rms = ", rms
      }
```

Type this program into a file called *meanrms* and type some numbers into an input file, perhaps called *meanrms.data.*

```
$cat >meanrms.data
20
30
55
40
<control-d>
$ awk -f meanrms meanrms.data
mean = 36.25
rms = 38.487
$
```

The **<control-d>** notation means you should press **CONTROL-**[D] to end the new file.

You don't have to create an input file to use this program. You can run it using the standard input (your keyboard):

```
$ awk -f rms_mean -
```

The dash for the input is optional.

Type in each number you want in the calculation, pressing [Return] after each entry. After typing the last entry and pressing [Return], type **CONTROL-**[D] (the end-of-file character). *Awk* then executes the *END* statement.

Because *awk* treats variables as strings until otherwise informed, giving character strings to this program produces unexpected answers. Mixing numbers and characters causes *awk* to calculate the mean and rms using the ASCII values of the characters.

Rearranging Data

You can use *awk* to change the format of records in a file. Assume you have this list of names in a file called *poets*:

```
Poe, Edgar Allan
Longfellow, Henry Wadsworth
Shakespeare, William
Frost, Robert
Dickinson, Emily
```

To transpose the first and last names, type

```
$ awk '{print $2, $1}' poets
```

This successfully switches around the first two fields, but it leaves out the middle names on two of the records and it leaves the commas on all the surnames.

To remove the commas, type

```
$ awk '{print $2, substr( $1, 1, length($1)-1 )}' poets
```

This complex-looking program drops the commas by printing a *substring* of the surname field consisting of everything *but* the comma, using the **substr** function. The first parameter, *$1*, is the object of the **substr** function. The second parameter, 1, means that we want the substring to start at the beginning of *$1*. The third parameter, *length($1)−1*, means that the substring should end one character before the end of *$1* (just before the comma).

Saving the middle names is more awkward. Try adding *$3* to the **print** action:

```
$ awk '{print $2, $3, substr($1, 1, length($1)-1)}' poets
```

The records with middle names come out correctly. But when no middle name is present an extra space occurs between the first and last names. The extra space is an output field separator being printed after *$3*; when *$3* is null (no middle name), you get two output field separators in a row.

One solution is to use an **if-else** statement to detect a middle name in the record. The program

```
{if (length($3) > 0) print $2, $3, substr($1, 1, length($1)-1)
else print $2, substr($1, 1, length($1)-1) }
```

when run:

```
$ awk -f reverse_names poets
```

results in:

```
Edgar Allan Poe
Henry Wadsworth Longfellow
William Shakespeare
Robert Frost
Emily Dickinson
```

Now that you have the format you want, save it by redirecting it to another file:

```
$ awk -f reverse_names poets >poets.awked
```

Another job you may want to do with a file like *poets* is to rearrange the records in alphabetical order. Unfortunately, *awk* cannot reorder records. Try using the *sort* utility. *Sort* is described in the *HP-UX Reference* and in the McGilton and Morgan book starting on page 138.

References

1. Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, "Awk—A Pattern Scanning and Processing Language", Bell Laboratories, September 1978. Second Edition. (Not available from HP.)
2. Henry McGilton and Rachel Morgan, *Introducing the UNIX™ System*, McGraw Hill, 1983, pp. 177-184. (UNIX is a trademark of AT&T Bell Laboratories.) HP Part # 98680-90025.
3. *HP-UX Reference for the HP 9000 Series 200/500.* HP Part # 09000-90006.
4. Brian Kernighan and Dennis Ritchie, *The C Programming Language*, Prentice-Hall, 1978. HP part # 97089-90000.
5. "C Library Routines," Programming Environment, HP-UX Concepts and Tutorials.

Index

a

b

Index (continued)

MANUAL COMMENT CARD

Text Editors and Processors
HP-UX Concepts and Tutorials

HP Part Number 97089-90022 10/87

Please help us improve this manual. Circle the numbers in the following statement that best indicate how useful you found this manual. Then add any further comments in the spaces below. **In appreciation of your time, we will enter your name in a quarterly drawing for an HP calculator.** Thank you.

The information in this manual:

Is poorly organized	1	2	3	4	5	Is well organized
Is hard to find	1	2	3	4	5	Is easy to find
Doesn't cover enough	1	2	3	4	5	Covers everything
Has too many errors	1	2	3	4	5	Is very accurate

fold

Particular pages with errors? _____

_____

Comments: _____

_____

_____

_____

_____

_____

_____

Name: _____

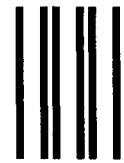
Job Title: _____

Company: _____

Address: _____

_____

Please Tape Here



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

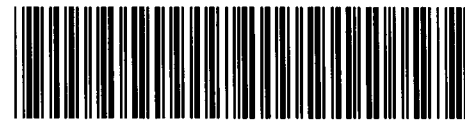
BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525

HP Part Number
97089-90022
Microfiche No. 97089-99022
Printed in U.S.A. 10/87



For Internal Use Only