ed Interactive Editor Tutorial

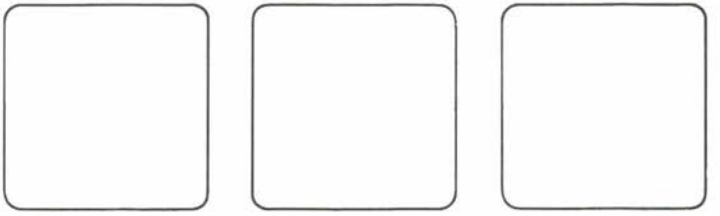

Table of Contents

1. Introduction

This is a User's Guide for the COHERENT interactive editor **ed**. It describes in elementary terms the facilities that the editor provides.

This guide is intended for two types of readers: those who want a tutorial introduction to **ed** and those who want to use specific sections as a reference.

Sections two through five cover the use of **ed**.

Section six gives a summary description of each **ed** command and its effects.

A related manual is the *Introduction to the COHERENT System*, which covers the basics of using COHERENT and introduces many useful programs.

Why you might need an editor

A significant feature of computers is the capacity to store, retrieve, and operate upon information. The kinds of information that can be stored by a computer running the COHERENT system are many: programs, computer commands and instructions, data for programs, financial information, electronic mail, natural language text (e. g. French, English) destined for a manuscript or book, or even notes to yourself.

ed is a COHERENT program that is designed to enter and change many kinds of computer-based information interactively. You will use **ed** to change computer programs and natural language manuscripts, command files, and electronic mail messages.

ed is designed to be as easy to use as possible, requiring little training to get started. The fundamental commands are simple, but have enough flexibility to perform complex tasks.

Learning to use the editor

Much care has been taken in the design and implementation of the editors and the writing of this manual. Practice on your part will help you learn quickly. The goal of this manual is to help you become proficient with **ed** as easily as possible.

This manual is designed to help you in the process of learning to use the editor. If there is someone learning with you, it is helpful for you to exchange notes while learning. Better yet, if there is someone who is an expert in **ed** whom you can talk to, it will help you gain familiarity more quickly.

Do the examples

The following sections will have many examples illustrating each topic of discussion. These examples are designed to assist you in understanding exactly how each command and feature will work.

Hopefully you have access to a COHERENT computer system. If you do, it is strongly recommended that you type in each example presented in the following sections as you encounter them in the text. Even if you understand the concept presented, performing the example will reinforce the lesson, and will help you feel comfortable using **ed** sooner.

If you do not have a COHERENT system available, take notes on the examples, showing what the results of each command would be. Then, when you do have access to a COHERENT system, go through the text and do the examples. You might find, to your surprise, that something new is revealed to you, even though you clearly understand the topic being presented.

Experiment on your own

In addition to reading the text and doing the examples as you encounter them in the text, feel no inhibition about trying a slightly different command than that presented in the example, and branching out on your own. Try things that you suspect might work, but are not shown as examples. Also try things that are a part of a project that you will be doing with **ed** on the computer, or that you are familiar with from previous experience.

Trying things out on your own is a good way for you to understand the editor in your own terms.

Interactive editing proficiency

As you grow in familiarity with **ed**, strive for an automatic use of commands—so that you do not have to look up each command, or laboriously think through the command you are about to type. Some commands, of course, are quite complicated, and require thought, but the commands that are used 90% of the time are simple. As you use **ed** more and more, the commands will come to you automatically. This proficiency will be helped by practice while you are learning **ed**. Learn to let your fingers do the thinking.

2. General topics

This section discusses the general ideas behind **ed**, and defines some basic terms. The topics discussed here will be referred to throughout the remainder of this User's Guide, and will be familiar to an **ed** expert.

To help illustrate the discussion to follow, log into your COHERENT system and enter the following commands:

```
ed
a
this is a sample
ed session
.
w test
q
```

This example calls **ed**, then uses the **a** command to add lines to the text kept in memory. The period signals the end of the additions. The **w** command writes the lines of text to file **test**, and the command **q** tells **ed** to return to the COHERENT system. You will notice that after you type the **w** command, **ed** will respond with

```
28
```

which is the number of characters in the file.

Thus, to enter **ed**, simply type

```
ed
```

and to exit, type

```
q
```

You can also exit by typing **<ctrl-D>** which is typed by holding the **control** key on your keyboard, then striking the **D** key. Notice that you are issuing two different kinds of commands in the above example. The command **ed** is a COHERENT command, while the rest are commands to the editor. After **ed** is given the **q** command, it exits, and following commands are processed by COHERENT.

Files

Sets of information stored by a computer are called *files*. In some ways, computer files resemble files in a typical office filing cabinet. All information in the computer is stored in files.

Each file has a name, which is used to refer to the file in COHERENT commands. The computer stores each individual file as a separate entity. The names of files are stored in a *directory*.

Each file name must be unique within a directory. Directories are discussed in more detail in the section on advanced editing. File names may be up to fourteen (14) characters in length. The characters that make up file names may be the upper case and lower case alphabet, numeric digits, and a few punctuation characters such as period and hyphens. The hyphen should not be the first character in a name, since many COHERENT programs treat file names beginning with hyphen incorrectly.

The COHERENT system has commands that create, destroy, list, copy and, with **ed**, enter and change files. For example, use the **cat** command to list the contents of the file named **text** created in the above example on your terminal:

```
cat text
```

ed and files

ed, like many of the other programs in the COHERENT system, deals with one file at a time.

You have control over the name of the file being created or changed. **ed** can create files, add to files, and change files previously created.

ed deals with files made up of *lines*. Lines contain upper and lower case alphabetic characters, the digits 0 through 9, and punctuation characters.

There are types of files that do not fit into this category. Such files contain computer instructions, or special program data. These are called binary files. The files **ed** deals with are called ASCII files.

Text and commands

As you use **ed** to create or change files, you will type both input *text* and controlling *commands* to the editor.

ed needs to be told what to do. You will use **commands** to order **ed** to do what you want it to do. **ed** has about two dozen commands, giving it power and flexibility.

The commands are almost always one letter and can be thought of as shorthand. Although the commands may seem over-abbreviated at first, they are easy to learn. You will appreciate the terseness of the commands once you begin to use **ed** regularly.

Each command to **ed** (and to COHERENT as well) is ended by striking the **<RETURN>** key. This key is present on all terminals. However, the labeling of the key may vary. It may be called **newline**, **linefeed**, **enter**, or **eol**, and is larger than any key on the keyboard except for the space bar. This key will be called the **<RETURN>** key in the remainder of this document.

The contents of a file is called *text*. The directions that you give **ed** are called *commands*. You will also enter text to fill and change the file. The commands tell **ed** what to do with the text.

Creating a file

ed operates upon one file at a time. **ed** will create a file with a name you supply and fill it with information.

The example shown in section two above created a file. Here is another example of file creation—**twoline**:

```
ed
a
Two line Example,
thank you.
.
w twoline
q
```

The letter **a** tells **ed** to add lines to the file. The file in this example is initially empty. The **w** command writes the lines you have added to file **twoline**. The command **q** tells the editor that you are

finished, whereupon it returns to COHERENT. You can use the COHERENT command **cat** to list the new file:

```
cat twoline
```

the reply will be:

```
Two line Example,
thank you.
```

Each individual command used here will be explained in detail in later sections.

Changing an existing file

Let's presume a manuscript file that you have created needs a few spelling corrections. **ed** will readily assist you in making the changes. Simply specify the name of the file when you issue the COHERENT command:

```
ed filename
```

where *filename* stands for the name of the file that you wish to change. To add another line to the example:

```
ed twoline
$a
This is the third line of the file.
.
w
q
```

Listing the program with **cat** gives:

```
Two line Example,
thank you.
This is the third line of the file.
```

The command **$a** tells **ed** to add lines at the end of the file.

The process of changing material in a file is frequently referred to as *updating*, or *editing*.

In your use of **ed**, you will issue commands to change, remove, and add to information in the file.

Adding information to a file is similar to creating a file. Information can be included in the current file from an already existing file.

Correcting the spelling of a misspelled word is easy with **ed**. Groups of words in an English manuscript may be rearranged.

Larger portions of text, such as a paragraph, may be moved or copied to a different spot in the manuscript.

Components of a file

Files that **ed** creates or edits are made up of ASCII characters. Commands also consist of ASCII characters, but some punctuation characters are significant to **ed** when used in commands.

Characters in the file are grouped into lines. A line is defined as a group of characters followed by an end-of-line character, which is not visible. **ed** operates upon the line as the basic unit of information; it is therefore a *line-oriented* editor. When you type out a file on your terminal, each line in the file will be shown on your terminal as one line.

Working on lines

ed knows each line in the file by its line number. The first line is known by the number 1, and successive lines by successive numbers. If your file has ten lines, the last line is known as number ten.

The commands for **ed** are based upon lines. When you add material to a file, you will be adding lines. If you remove or change items, you will do so to groups of lines.

The commands that you give to **ed** will be typed on a line of the screen. No part of any command that you issue will be acted upon until the line is completed. The processing will occur when you hit the **<RETURN>** key.

Referring to lines

As mentioned above, **ed** keeps track of the number of each line in the file you are editing. **ed** also remembers the line you most recently worked on. This can help shorten the commands you type, as well as reduce the need to remember numbers of lines. The line most recently worked on is called the **current** line. This phrase will be used in the following sections, and is frequently used by

experienced **ed** users. There is a shorthand symbol used in **ed** commands for the current line. It is the period '.' or dot.

Another shorthand symbol used in **ed** commands is **$** which represents the number of the last line in the file.

Line number ranges

Many of the **ed** commands operate on more than one line at a time. Groups of lines are denoted by a line number range, which is used as a prefix to the command. Line number ranges expand the power of **ed** commands.

Error messages

If you type a command to **ed** incorrectly, **ed** will respond with

```
?
```

signifying that an error has been detected. Many times, this error will be evident to you when you review the command that you just typed.

If you do not see what the error is, you can get a more lengthy description by typing to **ed**

```
?
```

and it will reply with an error message.

Summary

This section briefly describes the basic ideas important in using **ed**. The ideas will be discussed in detail and illuminated with examples in the sections that follow.

3. Basic editing techniques

This section discusses the elementary techniques and commands that you will need to begin using **ed**. With the material presented in this section, you will be able to perform operations needed to do most of your editing tasks.

Again, it is recommended that you follow along with the examples by keying them in. This will help you understand each example better, as well as remember the technique.

Creating a new file

To begin, let us presume that you need to create an entirely new file named **first**. Perhaps you only want one line in the file, and it is to read

```
This is my first example
```

These are the steps that you will need to go through to create this file.

The first step is to log into the system. If you do not know how to do this, then you need someone to help you with this step; see the *Introduction to the COHERENT System*. COHERENT will signal you that is ready for commands by typing a character called a *prompt*. This character is usually a **$**, but it may be a different character for some installations.

The next step is to invoke the **ed** program. To do this, simply type

```
ed
```

Remember that you must end each line of commands or text line with the **<RETURN>** key, for it will not be acted upon until you do. Thus, the editor is invoked by typing the two characters "ed" and a **<RETURN>**. Notice that these two characters must be lower-case. If you type either of them in in upper case, COHERENT will tell you that **Ed** or **eD** or **ED** is not found. Almost all COHERENT commands are in lower case. Always be sure the case of commands you type is correct.

ed is now ready for commands. The first command that you will use is the **a** command. This tells **ed** to append, or add lines to the text in memory, which will be later written to the file. Depending

upon the size of your computer, **ed** will hold only a certain number of lines of text in memory. For editing very large files, use **sed**, which is described in *sed Stream Editor Tutorial*. **ed** will continue to add lines until you type a line containing only a period. While adding lines, **ed** will not recognize commands.

Following the **a** command, type the lines to be included, followed by a line that contains only a single period. This special line signals **ed** that you want to stop appending lines. The information that you have typed so far is:

```
ed
a
This is my first example
.
```

Next, you must tell **ed** to write the file using a **w** command, and give the file name. If you wish to store this example in a file named **first**, issue the command

```
w first
```

which tells **ed** to write the information to the file named **first**.

ed will write the file and tell you how many characters were written, in this case, 25.

Finally, to leave the editor, issue the command

```
q
```

meaning *quit*. The next commands you type after this will be interpreted and acted upon by COHERENT.

Now review the example in its entirety. First, you called **ed**. Then you added lines with the **a** command, finishing the adding with a line containing only a period. You wrote the file with the **w** command, and exited from the editor using the **q** command. The complete example is:

```
ed
a
This is my first example
.
w first
q
```

ed replied to the **w** command with the number of characters written to the file. After you typed **q**, COHERENT prompted you for a command again with **$**.

Changing a file

Now, let's say that you wish to change the file that you have just created. You will add two more lines to the file so that the original line will be sandwiched between the new lines. You want the file to contain:

```
Example two, added last
This is my first example
Example two, added first
```

You will do this with **ed** using two new commands.

Again, you start by telling the COHERENT system to run **ed**. But this time, since you are changing a file, you type the name of the file that you are changing after the characters **ed**:

```
ed first
```

Notice that there is a space following **ed**. At least one space is required to separate the file name from **ed** itself. **ed** will remember this file name for later use in the **w** command.

ed reads the file in preparation for editing, and tells you the number of characters that it read in, again responding with 25.

After reading the file, **ed** automatically sets the current line to the last line read in.

Now, add the third line shown in the second example by entering:

```
a
Example two, added first
.
```

This resembles the first example. However, in that case there was no information in the file, whereas now there is. How did **ed** know where to add the lines?

The **a** command adds lines after the *current line*. And since upon reading the file **ed** sets the current line to the last line read in, the **a** command added the new line after the last line.

The current line can be implicitly or explicitly referred to by most commands, so it is helpful to know where it is. In general, the current line is left at the last line **ed** has processed. If you lose track of the current line, you can ask **ed** to tell you where it is, as you will see shortly.

To add the very first line to the second example, you will use yet another command, the **i**, or **insert** command. This command is similar to the **a** command, except that it inserts lines **before** the current line rather than after it. Otherwise, it is used to add lines in the same manner.

Another word about the current line. After an **a** command finishes, the current line is the last line added. Thus, after the addition of "Example two, added first" above, the (new) current line is the last line in the file. So, if you were to immediately do the **i** command, you would be adding lines just before the last line, which is not what you want to do.

ed has flexibility built into nearly every command to specify the line that the command is to operate upon. Now you can complete the second example:

```
1i
Example two, added last
.
```

The numeral **1** before the **i** says to insert lines before the first line in the file. The line number prefix is very frequently used, and is applicable to almost every command.

Now, to finish the second example and save it back into the same file, type:

```
w
q
```

Notice that the file name was left off the **w** command. **ed** remembers the file name that you started out with, and uses that name if none is given in the **w** command. Therefore, the information will be written back to file **first**. Notice also that the previous contents of file **first** are lost when you write the new file **first**. Alternatively, you can type

```
w second
```

leaving the contents of **first** unchanged and creating a new file called **second**.

In case you forget, **ed** will tell you what file name you started with. Simply use the command **f**

```
f
```

If you used command **f** anytime during work on this second example, **ed** would reply

```
first
```

Remember to use the **q** command to leave **ed** and go back to COHERENT.

Printing lines

As you work with a file using **ed**, it is most useful to print sections of the file on your terminal. This can help you see what you have done (and sometimes what you have not done) and help pinpoint where you wish to make changes.

The print command **p** will print the current line unless a line number is specified. Continuing with the example above,

```
ed first
p
```

ed replies by printing

```
Example two, added first
```

which is the last line in the file named **first** from the previous example.

Again, like **i** and **a**, if you want **ed** to print a line other than the current one, all you need to do is to put a line number or line

number range in front of the **p** command. Thus, if you want to print the second line in the file, type

```
2p
```

and **ed** will reply with

```
This is my first example
```

If you wish to print the entire example file, you can specify not just one line number but a range of line numbers to be printed. The first and last numbers are separated by a comma. So, to print all three lines in the second example, type:

```
1,3p
```

and **ed** will respond by printing all lines. This same principle applies to other commands. The print command can also appear after other commands such as **s** or **d**, which are discussed later in this section.

Abbreviating line numbers

There are shorthand descriptions for certain line numbers. The last line is frequently referenced, but since we don't always know how many lines there are, the number of the last line can be represented by dollar sign **$**. The command

```
1,$p
```

will print all lines in the file. The advantage of this shorthand is that the command as typed will work for any file, regardless of its size. This construct of **1,$** is used often enough that it has an abbreviation of its own:

```
*p
```

The number of the current line can also be abbreviated by using the period or dot in the place of a line number. To print all lines from the beginning of the file through the current line, type

```
1,.p
```

or to print all lines from the current line through the end of the file, type

```
.,$p
```

using two shorthand characters in the same command.

A special symbol **&** will print one screen of lines, which is useful if you are using CRT. Simply type

```
&
```

which is equivalent to

```
.,.+22p
```

unless there are fewer than 23 lines between the current line and the end of the file. In this case, it is equivalent to

```
.,$p
```

Be aware that all forms of the **p** command will change the current line to the last line printed. The command

```
.,$p
```

will, after printing, change the current line to the last line of the file.

How many lines

You can easily see the current line with **p**:

```
p
```

which is a very short way to tell **ed** to print the current line. On your terminal, try the command

```
.p
```

to see what it will do and how it compares to simply using **p**. You'll see that they do the same thing.

You can determine the size of your file by typing

```
=
```

ed will reply by typing the number of lines in the entire file.

To determine the line number of the current line, use the **dot equals** command:

```
.=
```

ed responds with the number of the current line.

Removing lines

An old saying says that what goes up must come down. In computer systems, that might translate to: "that which is remembered may also be forgotten". **ed** helps you forget lines the morning after, or even sooner if you wish.

To illustrate the removal of lines, let's create another example file with **ed**:

```
ed
a
This is the first line.
The second line is good.
However, line three is bad.
line four wishes to go away.
line 5 similarly wants to be forgotten,
as does line 6.
the next to last line stays.
as does the last line in the file.
.
w example3
q
```

This will create file example3. You can remove lines that you don't want from this file.

To delete the lines, begin editing the file by saying

```
ed example3
1,$p
```

This also prints the file on your terminal. Now, your intent is to delete lines three through six. First, delete line three, then print the entire file again.

```
3d
1,$p
```

and **ed** will respond with

```
This is the first line.
The second line is good.
line four wishes to go away.
line 5 similarly wants to be forgotten,
as does line 6.
tne next to last line stays.
as does the last line in the file.
```

Notice that the third line is no longer there. Line three is now what used to be line four. Remember that the line numbers *always* begin at one for the first line of the file and progress consecutively even after the file has been changed. Thus, deleting a line will change the line number of each line from the deleted line to the last line in the file.

Your deleting is not finished, however. You need to remove three more lines. This can be done with one command:

```
3,5d
1,$p
```

Again, **p** will print the contents of the file, which now are

```
This is the first line.
The second line is good.
the next to last line stays.
as does the last line in the file.
```

Finally, write the updated file and return to COHERENT:

```
w
q
```

This illustrates how to delete lines, both singly, and in a group.

Abandoning changes

If you should make an inadvertent deletion or two and wish to start the edit over again from the beginning, you can do so by using the **q** command in a different fashion than is shown above.

If you tell **ed** to **q** before you tell it to write the file with **w**, you can abandon any changes made since beginning editing. However, to prevent you from accidentally selecting this option, **ed** will respond

with a question mark "?" if you have made any changes to the file. At this point, reply with a second **q**, and **ed** will then return to COHERENT.

ed is cautious about letting you quit when you have made changes that have not yet been written to the file by **w**, so it requires that you do the **q** twice in this manner. Alternatively, you can avoid the question mark prompt by typing the upper-case **Q** rather than lower-case **q**, and **ed** will exit without regard to unsaved changes.

You can also exit from **ed** by typing the end of file key, which is usually **<ctrl-D>**.

Although you are keying changes to the file as you go along, the file is not permanently changed until you issue the **w** command. These modifications are made on a copy of the file text held in memory.

Changing text within a line

If you type a line incorrectly, or later wish to rearrange some words or symbols within it, you know enough about **ed** now to do so. You only need to delete the line with **d** and re-enter the line with **i**. Example 4 will be created with the following commands:

```
ed
a
Software technology today has
adbanced to the point that large
software projects unherd of in
earlier times are undertaken and
.
w example4
q
```

There are two misspelled words in this example and we will correct each of them using different **ed** features.

The first method will be the direct way that you probably can anticipate. Give the following commands to the editor exactly as shown:

```
ed example4
2d
i
advanced to the point that large
.
```

These commands replace the second line with a new line containing the correct spelling of the word **advanced**. Use the command

```
1,$p
```

to verify that the file now will contain:

```
Software technology today has
advanced to the point that large
software projects unherd of in
earlier times are undertaken and
```

The second method used to change the spelling of a word is with the substitute command **s**. This command is very powerful. It is probably the most-used command in **ed**.

s is more complex than commands we have discussed so far, in that there are more elements to the command. First is the optional line number range followed by the **s**. Then there are two *patterns* or *strings* that are set off from the rest of the command and from each other with the **slash** character:

```
s/pattern1/pattern2/
```

In this example of the substitute command, the string *pattern1* represents the word or string that you want **ed** to find, then change. The string *pattern2* is the word or string that is the replacement for *pattern1*. Notice the three slashes separating the two patterns from the **s**, from each other, and from the end of the line. These slashes must always be present.

With this command, you can correct the second spelling error in the fourth example:

```
3s/herd/heard/
p
```

ed will reply

```
software projects unheard of in
```

Notice that these two command lines can be condensed to one:

```
3s/herd/heard/p
```

The meaning of these commands is: on the third line of the file, change the string **herd** to **heard** and, when finished, print the entire line. Without the **p** above, **ed** will change the line as you direct, but you will not see what the new line is. It is a good idea to print lines that you substitute in this manner until you gain in confidence with **ed**. Some **ed** experts always print the lines after substitution.

After these two changes, the file will look like:

```
Software technology today has
advanced to the point that large
software projects unheard of in
earlier times are undertaken and
```

Although the above example is based on patterns in the s command as words, they can be any consecutive group of characters, called *strings*. Either pattern may be several words, or part of a word. **ed** really doesn't know what words are, but it does know what arbitrary strings of characters, or *patterns*, are.

Because **ed** is not strictly examining words, you should keep in mind that it may find the wrong *pattern1* string on the line in question. The substitute command finds the first *pattern1* on the line that matches. For example, presume that the current line in a file is

```
let not rain fall on a parade
```

and instead you want to say

```
let not rain fall on the parade
```

you command **ed**:

```
s/a/the/p
```

and are shocked to discover that the result is

```
let not rthein fall on a parade
```

which is certainly worse than what you started with. A better command to give **ed** would have been a substitute command that substituted the letter **a** preceded and followed by a space:

```
s/ a / the /p
```

Notice that will find only one " **a** " with this command.

An alternative correct way to do this is to indicate in the substitution command which of several possible matches within the line is to be substituted. In our example, it is actually the third **a** that we are trying to match, so we could have used the special form of the command

```
s3/a/the/p
```

to get **ed** to select the one we wanted.

Undoing substitutions

If you did change **a** to **the** inappropriately, you can retract the substitution by issuing the undo command

```
u
```

before you move on to another current line.

To illustrate this, enter this example:

```
ed
a
let not rain fall on a parade
.
w undo
q
```

Now, perform the substitution with

```
ed undo
s/a/the/p
```

which will result in:

```
let not rthein fall on a parade
```

To retract the substitution, simply type:

```
u
p
```

to undo it, and print the result.

Note that only the last line substituted will be restored, and it must still be the current line.

The **s** command finds only one occurrence of the string that you want to change, so if there is more than one misspelling of a word on the same line, you would need to give the command twice.

However, there is a different form of the substitute command which will find every occurrence of the indicated string on the line. Simply add the letter **g** for **g**lobal after the third slash in the substitute command, and every one will be found and changed:

```
s/pattern1/pattern2/g
```

So, if the current line contains a phrase:

```
a rose is a rose is a rose
```

and we tell **ed** to substitute

```
s/a/the/g
```

the line will be changed to

```
the rose is the rose is the rose
```

Again, be wary of the wrong word or part of a word inadvertently matching the string that you want to change.

There are some special punctuation characters that the substitute command uses in parts one and two. They will be discussed in the advanced section of this document. However, you should be aware of these characters and avoid them until you progress to the advanced section, for unless used properly, they will give you undesired results. These characters are:

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

These are used in **ed** and other COHERENT programs in forming complex patterns.

Ranges of substitution

Perhaps you need to change several lines that have the same misspelling or need the same editorial change. **s** can do that for you also. Simply prefix the command **s** with the line number range like you would do with **p**. Borrowing the "rose" example again, if the saying were typed:

```
a rose is
a rose is
a rose
```

then you could do the same change as before, but across the entire file by typing

```
1,$s/a/the/
```

Notice that the **g** following the **s** command has been omitted here, since you know that there is only one occurrence of the string that you want to change on each line.

If some of the lines do not have the string you want to change in them, **ed** will not object to the missing string. However, if none of the range has the string, **ed** will print a **?**.

Summary

This section discusses the elementary commands essential for you to begin basic editing. Later sections will cover additional flexibility in these commands, as well as demonstrate more powerful commands.

You can build a new file with the command sequence

```
ed
a
<lines to be added>
.
w filename
q
```

and edit an existing file with the command

```
ed filename
<editing commands and text>
w
q
```

To print lines of a file, you use

```
p       {print current line}
np      {print line n}
m,np    {print lines m through n}
```

and either line number may be replaced by **$** signifying last line or period signifying the current line.

To add lines to a file, use

```
a       {add lines after current line}
na      {add lines after line n}
```

The command **i** is similar, except that it adds *before* the indicated line.

To remove lines from the file, use

```
d       {delete current line}
nd      {delete line n}
m,nd    {delete lines m through n}
```

And finally, the substitute command

```
s/p1/p2/        {change first p1 to p2 in cur. line}
sn/p1/p2        {change nth p1 to p2}
s/p1/p2/g       {change all p1 to p2}
m,ns/p1/p2/g    {ditto on lines m thru n}
```

The substitute command will give an error message if no *p1* is found—that is, at least one *p1* must be present in the indicated range.

4. Intermediate editing

This section discusses more advanced command features of **ed**. While section three discussed enough material to help a first-time user become productive, this section covers additional features that can considerably increase editing power.

The topics covered in this section are: relative line numbering, moving blocks of text, string locating, special characters in substitution and search commands, global command processing, marking lines, and reverse searches.

Examples of each command are given.

Relative line numbers

As discussed in the previous section, most commands accept line numbers to control their range of operation. The line number specification may be a single number before the command, such as:

```
1p
```

which, of course, prints the first line of the file. The line number specification may also be a range of line numbers, indicated by two numbers separated by a comma. If the file has at least ten lines in it, the command

```
1,10p
```

will print the first ten lines of the file.

You may specify the current line by simply using dot to represent the current line number, as in

```
1,.p
```

which will print the lines of the file up through the current line. If you want to refer to the current line only, you may omit the line number prefix altogether, as in

```
p
```

which is in every way equivalent to

```
.p
```

but is shorter.

There is yet another level of shorthand of the print command—the plus and minus characters. These characters can be used to indicate offsets from the current line as in

```
.+3p
```

which means to print the third line following the current line.

```
.-1p
```

means print the line preceding the current line.

This may be abbreviated further by leaving out the dot. The command sequence

```
+p
-p
```

will have the cumulative effect of advancing to the next line as the current line, printing it, then backing up to the previous line (the original current line) as the current line and printing it.

Further, you can put several of these on one command line to move the current line multiple lines, then print. To back up three lines then print, say:

```
---p
```

In the absence of any other command, **ed** defaults to the **p** command. Thus

```
---
```

is equivalent to

```
---p
```

and

```
5
```

means the same thing as

```
5p
```

There is one more abbreviation in the print command.

If **ed** is expecting a command from you, and you enter nothing except a **<RETURN>**, **ed** interprets this as a command to advance

the current line to the next line and print it. How about that for brevity! This action is equivalent to

```
+
```

or

```
.+1
```

<RETURN> is the shortest command in **ed**.

All the abbreviations for line number can be used by other commands that expect a line number range. For example, if you want to delete five lines centered about the current line, you could type:

```
.-2,.+2d
```

and you would get your wish.

With any of these abbreviations, as well as the specification of the actual line number itself, you may not specify a line number that is beyond the limits of the file. Suppose the current line is the last line in the file and you type a

```
+
```

to **ed**. This means *advance one line then print*, which cannot occur, since there is no next line in this case. **ed** will respond to improper line numbers by typing a question mark on the terminal. Notice, however, that the current line will always be valid so long as there is at least one line in the file. Thus, unless the file is empty, the command

```
.
```

will never give an error message. This can be of comfort if you lose your way in the file.

Changing lines

In the Basic editing section, an example of spelling correction was solved two ways. The first way was the clumsy way of deleting a line and retyping the entire line. Such an activity is a lot of work to change a single letter, so the substitute command was used instead.

There are occasions, however, where it is handy to have the power to change lines—as was done by deleting then inserting. **ed** provides this power in the **c** command. In general terms,

```
m,nc
new lines
to be inserted
.
```

will remove lines *m* through *n*, and insert new lines up to the period in place of them.

Moving blocks of text

In a natural language manuscript, you often need to rearrange paragraphs to give better clarity. In a program, procedures may need to be rearranged. Or, possibly you forgot where the current line was, and inserted lines not quite where you wanted them.

ed provides a move command **m** that moves a block of text from one point in the file to another.

The analogue to this operation in a conventional typed manuscript is to cut out the section from the wrong place, move it to the new place, and paste it in.

m is different from the other commands that we have discussed so far, in that there is a line number following the **m** itself, as well as the line number range that normally precedes a command. The following line number is interpreted as the line after which the text is to be moved. So, the general form of the move command is

```
b,emd
```

which means move lines *b* through *e* to follow line *d*.

To do a concrete example, build a file with the following information:

```
ed
a
    This is a paragraph of natural language
text.   Due to stylistic considerations, it
really should be the second paragraph.
    If you can read this paragraph first,
the text has been properly arranged, and
our move example has been successfully done.
.
w example5
q
```

The file **example5** is a section of a manuscript with two paragraphs of three lines each. The purpose of this example is to move the first paragraph to follow the second paragraph. There are at least two ways to do this with the move command. The first is

```
ed example5
1,3m$
*p
Q
```

The result will be

```
    If you can read this paragraph first,
the text has been properly arranged, and
our move example has been successfully done.
    This is a paragraph of natural language
text.  Due to stylistic considerations, it
really should be the second paragraph.
```

This example moves the paragraph at lines one through three to the end of the file (**$**). The other way is to move the second paragraph to the point before the first:

```
4,6m0
```

Notice that the destination is 0, meaning that the text is to be moved to the point following line zero. Since there is not a line number zero, the move command takes it to mean the beginning of the file.

Of course, with our small example, there are several other ways using line number abbreviations and knowledge of the current line to perform exactly the same action.

```
1,3m.
```

say to move lines 1 through three of the file to the line after the current line. Immediately after the **ed** command, the current line will be the last line of the file. Thus, this form of the command has the same effect as the previous forms.

If the destination of a move command is not specified, **ed** assumes the current line. Therefore, the command

```
1,3m
```

will also have the same effect.

In this discussion about the move command the resulting current line in comparing the different ways of performing the task has not been mentioned. The different methods are equivalent with respect to the resulting order of lines after their execution, but not necessarily the same with respect to the new current line. The **m** command causes line numbers in the file to be changed, although the total number of lines in the file remains the same.

After a move command, the current line is defined to be the last line moved. Thus, if the first paragraph is moved, the current line after the move will be original line three, now the last line in the file. If the second paragraph is moved, the current line after the move will be the new line three.

Copying blocks of text

The transfer command **t** is similar to the move command, except that the text is copied rather than moved. The term *move* when applied to lines of text generally implies that the moved object no longer occupies its original place.

ed adheres to this meaning when you command it to move lines of text. The term *copy* however, generally means to move a copy of an object, such as a block of text, but leave the original in place. **ed** interprets the transfer **t** command in this fashion.

The form of the transfer command is:

*b,e*t*d*

which means to transfer the group of lines beginning with *b* and ending with *e* (inclusive) to follow the line *d*.

This would be used if you have a paragraph in a manuscript that bears repeating. The original section of text is not altered.

After copying lines to the destination, **ed** sets the current line to the last line copied.

String searches

As if we did not have enough ways to refer to lines, there are still more to come!

The methods discussed to this point are the simplest to understand and to use. They involve specifying an absolute line number, a relative line increment, or a shorthand symbol such as **dot** or **$**.

Particularly in a natural language manuscript, line numbers are a bit arbitrary, in that there is no intuitive grasp of which line has which number, how many lines ago a paragraph starts, and so on.

ed's solution to this is a string search or *line locator* capability to locate lines, using a syntax resembling the substitute command. The string search begins on the line following the current line, and looks for a line matching the specified string. If a match is found in a line **ed** sets the current line to that line.

If the end of the file is encountered before a match is found, the string search continues at the beginning of the file. If there is still no match by the time the beginning line of the search is encountered, **ed** will issue an error message—the question mark **?**. Remember that if you answer **ed** after an error message with a question mark, it will tell you in more detail what the error is.

What does a match mean? The simplest meaning is that two strings are the same—the strings have the same characters in the same order.

Build an example by typing the following lines:

```
ed
a
    This is an example that we will
use for string searching.  There
is much natural language here as well
as some genuine arbitrary strings.
890,;+   foxtrot
qwertyuiop ##
.
w example6
q
```

Of course, these lines can be referenced by the means already discussed. However, if the file being edited contains fifteen typewritten pages of information, these methods become impractical.

The string search is a method of locating a line. You can place the string search command at any place that you would place a line number or line number expression. To illustrate the action of the string search, let's locate any line with the word or partial word **fox** and print it.

```
ed example6
/fox/p
```

When you type this line, **ed** will print the line

```
890,;+   foxtrot
```

Also, you can print out a range of lines using the string expressions:

```
ed example6
/This/,/much/p
```

will print out the lines:

```
  This is an example that we will
use for string searching.  There
is much natural language here as well
```

This ability to specify strings as targets for locating lines gives a greater power to **ed** overall. Once you get the feel of this feature, you will begin to see the true power and flexibility of **ed**.

The searches can also enter into relative line number expressions. If you have a Pascal program file with several procedures in it, but you find that you need to rearrange the procedures, you can combine the power of the move command with the string searches.

```
PROCEDURE A;
...
...
PROCEDURE B;
...
...
PROCEDURE C;
```

Presume that the section of text beginning with **PROCEDURE A** needs to follow the line containing **PROCEDURE B**. The following move command will do the move properly:

```
/PROCEDURE A/,/PROCEDURE B/-1m/PROCEDURE C/-1
```

This commands **ed** to move the section of the file beginning with the line containing **PROCEDURE A** and ending just before the line containing **PROCEDURE B**. This section contains procedure A. These lines are to be moved before the line containing **PROCEDURE C**.

Let's explore this in a bit more detail. Remember that the move command is defined as

b,*e*m*d*

where *b* indicates the first line of the text to be moved, *e* indicates the last line of the text to be moved, and *d* indicates the line that the moved text is to follow. Thus, *b* corresponds to the number of the line containing **PROCEDURE A** and is the first line of the procedure in question. But *e* corresponds to the line before (by virtue of the −1) the **PROCEDURE B** begins. This line must be the last line of the A procedure. Thus, you have found the beginning and ending lines of procedure A.

The final string search locates the first line of subroutine C. The move command expects the *d* to be the line that precedes the moved text, and so we must subtract one from the line number of the string **PROCEDURE C**.

Thus, you can use a string search anywhere that you are allowed to use a line number, even in a relative line number calculation. While this example may appear a bit involved, remember that it is but a compact way of describing how to find the beginning and end of the desired text, as well as the location that it is to be moved to.

Practice with the string searches; if you master them well, they will be powerful tools for you.

Remembered search arguments

As discussed earlier, line number abbreviations may take many forms. They may be entered as ., or +, or −, and certain combinations of these. In some commands, no line number entered means the current line number is to be used.

ed encourages abbreviation in the search string. If no string is entered between the slashes in a search or substitution (or question marks) then **ed** takes this to mean that it is to use the last-used search string. A common use is found in a global substitution command (which will be discussed in detail later in this section).

```
g/please remove this string/s// /p
```

which does not quite remove it, but replaces it by a blank. The last-used string can be specified by a string search, a substitute command, or a reverse string search (also discussed later in this section). Also, the remembered search argument may also be used in any one of these. You can use the remembered search feature to "walk" through the file, finding the next occurrence of a remembered search pattern.

Uses of special characters

As powerful as the line locator seems, there are even more powerful features. These will be discussed in the Expert editing section below.

However, these more powerful capabilities depend upon certain punctuation marks used in a special way. As you use the line locator (as well as the substitute command) be aware of these following characters:

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

for they have special significance to **ed** when appearing in a string search or a substitution pattern.

If you need to use one of these characters without invoking its special meaning, precede the character with a backslash '\'. This tells **ed** not to interpret the character in a special way.

For example, to find a backslash character, type the search command:

```
/\\/
```

If any of these characters is to be used in another context, say within lines that you are adding with the **a** command, it should not be preceded with the backslash. Only use the backslash to hide the meaning when it appears within the string search command, or within the first part of the substitution command.

Global commands

The global commands **g** and **v** give you the capability to repeat commands on all lines within the specified range that contain certain strings. For example, to print all lines that contain the word **example**:

```
g/example/p
```

The global command may be a prefix to almost any command. The following command will delete all lines that contain three consecutive plus signs:

```
g/+++/d
```

while the command

```
g/foxtrot/.-2,.+2p
```

will print the five lines surrounding any line containing the word **foxtrot**.

A very common use of the global command is to perform global substitution. The command

```
g/PROCEDURE/s/PROCEDURE/PROC/gp
```

will perform the substitution on each line that contains the string **PROCEDURE** and print the resulting line.

This may appear similar to the command

```
1,$s/PROCEDURE/PROC/gp
```

but is different in that the global command will print each of the changed lines, while the substitute command will print only the last line changed. Also, the method of operation of these two commands is different.

A related command **v** performs much the same task, but will execute the commands only for lines that do *not* contain the specified string. Thus, to print all the lines that do not have the letter **w**, use

```
v/w/p
```

For more sophisticated uses of the **g** and **v** commands and how they work, see the section on Expert editing.

Joining lines

What do you do if you inadvertently hit **<RETURN>** as you are adding lines and need to combine the two lines?

```
ed
a
Look out, I seem to have hit ret
urn in the
middle of a word and don't know
what to do!
w rid
q
```

Rather than retyping the entire line, you can use the join command **j**:

```
ed rid
1,2j
1,$p
```

will give

```
Look out, I seem to have hit return in the
middle of a word and don't know
what to do!
```

If no line number is specified, **j** will join the current line and the following line. If a single line number is specified, join will operate on that and the following line.

Several lines can be joined by using the form of the command

a,*b*j

where lines *a* through *b* will be joined into a single line. The command

1,$j

will join all the lines in the file into a single line. Then, the command **.p** or **p** will print the entire file.

Notice that the command

3j

does the same job as the command

3,4j

The join command generates its own second line number if none is specified, so that the command

*n*j

is equivalent to

n,*n*+1j

where **n** is a line number. This command is the only command that interprets a missing line number this way.

Splitting lines

You can split one line into two with the substitute command. To illustrate, suppose you typed in the following commands:

```
ed
a
This line wants to be two, with this second.
.
w split
q
```

To perform the split, type

```
ed split
s/two, /two,\
/p
*p
wq
```

The line split is caused by the backslash preceding the **<RETURN>**. This tells **ed** that the **<RETURN>** does not terminate the command, but that the **<RETURN>** is part of the substitution. The contents of file **split** are now

```
This line wants to be two,
with this second.
```

Marking lines

As you are editing a manuscript or program, it is sometimes handy to be able to leave a bookmark in the text for easy later reference. **ed** provides this feature by means of the mark command **k**. To mark the next line that has the word **find**, use:

```
/find/ka
```

where the letter **a** is the mark. To later print the line that has been so marked, use:

```
'ap
```

These references may be placed anywhere that a line number is expected.

The mark must be a single lower-case letter. Also, each mark will be associated with one line. Marking a line with the **k** command does not change the current line.

The use of marks can be especially handy in moving paragraphs with the **m** command. Using marks can give you a chance to review the sections that you will be moving before you do the move.

Let's say that you have a manuscript with a paragraph that needs to be moved to a different part of the document. Create the following example:

```
ed
a
    This is a paragraph, first line, that
needs to be moved.
text
text
And this is the last sentence of the paragraph.
    Next paragraph begins here.
text
text
text
    This is the spot that we want the paragraph
to precede.
.
w example7
q
```

Now, let's place three marks to help with the move:

```
ed example7
/first line,/ka
/Next paragraph/kb
/is the spot/kc
```

This marks the first line to be moved with **a**, the line following the last to be moved with **b**, and the paragraph's destination with **c**. But you can see that the move command moves lines to the line *after* the third number specified, so let's change the third mark:

```
'c-1kc
```

so that we can use **c** in the move command without arithmetic. Now, print the paragraph to be moved to be sure that the marks are correct.

```
'a,'bp
```

ed will reply with

```
    This is a paragraph, first line, that
needs to be moved.
text
text
And this is the last sentence of the paragraph.
    Next paragraph begins here.
```

You can see that we would move one line too many if we used the marks as they are. So, let's change **b** also.

```
'b-1kb
```

Now, we can do the move:

```
'a,'bm'c
1,$p
```

The file will now contain:

```
    Next paragraph begins here.
text
text
text
    This is a paragraph, first line, that
needs to be moved.
text
text
And this is the last sentence of the paragraph.
    This is the spot that we want the paragraph
to precede.
```

Marking sections of text can increase the ease with which you solve your complex **ed** problems.

Searching in reverse direction

All scanning, processing and searching has been shown going from the beginning of the file towards the end. Sometimes it is useful to find some word that occurs **before** the current line.

You can get **ed** to do string searching in the reverse direction by specifying the search with question marks **?** rather than slashes /. To find the previous occurrence of the word **last**, use:

```
?last?
```

This manner of searching can be useful in finding the beginning and end of a **repeat/until** statement, for example. If the current line is in the middle of a Pascal **repeat/until** group, you can print the group with the command

```
?repeat?;/until/p
```

The reverse search is like the forward search in every way except the direction of search. The search begins one line before the current or specified line, and proceeds toward the beginning of the file. If the string is not found by the time that the search reaches the beginning of the file, the search resumes at the end of the file, and progresses towards the starting point of the search. If the string is not found by the time that the search reaches the original starting point, the question mark error message is issued signifying no match.

Also, the command

```
??
```

will use the remembered search argument.

Summary

This section covers intermediate **ed** topics, building on the ideas and features presented in the basic section.

ed accepts many alternative forms of line numbers, from absolute line numbers through the very shortest abbreviation. Those forms are reviewed here, with descriptions enclosed in the braces { and }.

```
1p      {print line one}
1,.p    {print from beginning of file to current line}
p       {print current line}
.p      {print current line}
+       {advance one line; print}
-       {back up one line; print}
+3      {advance three lines; print}
---     {back up three lines; print}
        {<RETURN> only, advance one line; print}
.       {print current line}
```

You can move blocks of text with **m** from one section of the file to another. Copying blocks of text with **t** is similar to moving them.

ed also has more powerful ways of line location, notably string searches. Examples show how this feature enables powerful use of the other features of **ed** while freeing you from the necessity of keeping track of all relevant line numbers.

ed treats certain characters in a special way. These characters are introduced, and this section shows you how to avoid unwanted side effects while using them.

Next, global string searches are introduced, along with hints on how to increase the power of other commands when used in conjunction with them.

Line joining, while not heavily used, can be very difficult to get along without in circumstances where it is needed. The join command is discussed, along with examples.

To make line referencing even easier, **ed** has a mark command **k**, and the capability to refer to marked lines. Both are discussed, along with how they might be used for more complex text manipulation problems.

Finally, the reverse string search feature is demonstrated.

5. Expert editing

This section presents the most advanced **ed** commands.

File processing commands

The Basic editing section discussed the COHERENT commands

```
ed
```

and

```
ed filename
```

There are additional file handling commands in **ed** that go beyond the power of those already discussed.

If you decide that you were editing the wrong file, or have finished the current file with a **w**, you may begin editing an entirely new file with the command:

```
e newfile
```

This forgets all the changes that you have made, if any, up to this point since the last **w** command and begins all over again with **newfile**.

The **e** command has the same effect as the COHERENT **ed** command with a file name:

```
ed new
```

issued to COHERENT is the same as

```
e new
```

issued within **ed**, but the second is handier since you do not need to exit **ed** then reenter to edit a new file. Note that the **ed** command **e**, like the **q** command, will issue an error message if another file is being edited and you have not stored it since your last change was made. If you immediately repeat the command, **ed** will go on even if there are unsaved changes. If you use the command

```
E new
```

ed will edit the new file, whether or not there are unsaved changes.

The **r** command also reads a new file, but adds the lines from it to the work in progress on the current file instead of destroying the

current file. This can be handy for including one file in another one. If you have a manuscript prefix stored in a file **prefix** and are editing a new manuscript, to include the prefix at the beginning, say

```
0r prefix
```

where **r** reads in lines of the file after the line number specified, or in this case, line zero, which means at the beginning of the file. Without a line number, **r** reads in lines at the end of the file.

The **w** command writes out the entire file if no line number is specified, but line number selection can be supplied.

```
1,3w new
```

writes out the first three lines to file **new**. If the file name is omitted, the lines are written to the remembered file name.

The **w** command is unique in that it never changes the current line. This is true regardless of what line numbers are specified in the range for the command, or how those line numbers were developed.

The **W** command is similar to the **w** command except that **W** appends lines to the end of the file, while **w** creates a new file, destroying any previous contents.

The **f** command prints the remembered file name that was set in

```
ed filename
```

or

```
e filename
```

or

```
w filename
```

commands. But **f** can also be used to set the remembered name by saying:

```
f newname
```

This form of the command will tell you what the new remembered file name is, even though you just typed it in.

Note that the command

```
w filename
```

changes the remembered name only if there is currently no remembered name, as does the **r** command.

Patterns

In earlier sections of this document, you were cautioned about certain punctuation characters having special effect in search and substitute commands. These characters are

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

and are used to form powerful substitute and locator commands. The combination of these special characters in an orderly way is called a *pattern*, sometimes called a *regular expression*. Patterns can specify a search string that can find or *match* a variety of strings with a single search argument.

The idea of patterns is based upon mathematics. These patterns are a particularly good way of describing general classes of strings.

The simplest patterns use alphabetic characters and numeric digits, which match themselves, as in

```
/ab/
```

which finds and prints the next line containing the string **ab**.

Matching any character

The next simplest character to use in a pattern is the period or dot. It will match any character except the **newline** character that separates lines. Two periods in succession will match any two consecutive characters, and so on. For example, if you have a file containing algebraic statements of the form

```
a+b
c+e
a-b
a/b
d*e
```

and wanted to find and print any line involving **a** and **b** (in that order), then the search statement

```
/a.b/
```

will do the trick. The . in this example will match +, −, and /.

Then, you ask, how do I find a string that contains a period? For example, if you wanted to find all the sentences that ended with "lost.", (that is, the word **lost** followed by a period) you might first try

```
/lost./p
```

but you can see that this would match the string "lost " (the word *lost* followed by a space) as well, which is not what you want.

This is where the special character backslash comes in handy. The function of the backslash is to tell **ed** that the next character is to be treated as a regular character, even if it is a special character. Thus, to find "lost.", you need only type:

```
/lost\./p
```

and you will not incorrectly find "lost ". And, if you want to find backslashes in your file simply say

```
/\\/p
```

Matching many of one character

ed will help you match strings that contain any number of repetitions of a specific character with the *. For example, to remove extra spaces between words in a document, type

```
g/##*/s//#/p
```

(The character # has been substituted here for the space character to make the example more readable.) This will replace each series of spaces by a single space.

Notice that there are two spaces before the * in the search string. This is necessary because the * will match any length of string, including zero. Therefore, searching for a space followed by any number of spaces will find strings that are at least one space long.

The * matches the longest possible string of the previous character. This will require careful attention on your part, since the string matched by * might be longer than your required string, or even zero in length. Either way could give you unexpected results.

If you have a line

```
a+b-c
```

in your file and want to change it to

```
a+c
```

type the command

```
s/a.*c/a+c/p
```

However, if the line read instead

```
a+b-c*d+c
```

and you applied the command, the result would be

```
a+c
```

since the **.*** will match the longest string between any **a** and any **c**.

Beginning and ending of lines

The characters ^ and **$** will match the beginning and ending of lines for you. Thus, you can find and print all lines that end with a bang:

```
g/bang$/p
```

Or those that begin with a whimper:

```
g/^whimper/p
```

These two characters can help you find lines of specific length, also. If you need to see all lines of exactly five characters in length, the command

```
g/^.....$/p
```

will do the trick. To find and delete all blank lines, do

```
g/^ *$/d
```

Notice this time the * will match a string of zero spaces. But this is correct, since a blank line includes lines that have nothing in them, as well as lines that contain only spaces.

Replacing matched part

In many cases of substituting, you find yourself extending a word, or adding information to the end of a phrase. This can lead to extensive retyping of characters. The special **&** character can help out.

This character is special only when used in the right part, or *pattern2* of the substitute command. It means "the string that matched the left part". For example, to add **ing** to the word **help** in the current line, use:

```
s/help/&ing/
```

The ampersand may appear more than once in the right side.

This can be more interesting if the left part has a non-trivial pattern. For every word in a line that is preceded by two or more spaces, double the number of spaces before it:

```
s/###*/&&/gp
```

(Again, spaces have been replaced with # for clarity.)

Replacing parts of matched string

A more sophisticated feature similar to the ampersand helps you to rearrange parts of a line. For example, create a file by typing

```
ed
a
first part=second part
.
w eql
q
```

There are two special bracket symbols, **\(** and **\)** that are used to delineate patterns in the left part of a substitution expression. Then, the special symbols **\1**, **\2** and so on, will be used to insert the delimited parts. The symbol **\(** marks the beginnning of the pattern, and **\)** marks the end.

To delete everything in the line except the characters to the left of the =, type

```
ed eq1
s/^\(.*\)=.*/\1/p
wq
.
```

In the substitute command, the ^ matches the beginning of the line, then .* will match "first part", and =.* will match the rest of the line. The symbol **\1** signifies the matched characters between the first **\(** (the only one in this example) and **\)**. The **p** prints the result, which will be

```
first part
```

With this example, you can interchange parts of a line:

```
ed
a
first part=second part
.
w eq12
q
```

To interchange the two parts, type

```
ed eq12
s/\(.*\)=\(.*\)/\2=\1/
p
wq
```

The result is

```
second part=first part
```

The first part of the substitution expression

```
\(.*\)=\(.*\)
```

can be thought of as being in three parts. The first part

```
\(.*\)
```

matches all characters up to but not including the =, which are

```
first part
```

The second part

```
=
```

matches the = in the line, and finally the third part

```
\(.*\)
```

matches all characters following the "=", or

```
second part
```

The remainder of the substitution expression

```
\2=\1
```

which is the replacement part, rebuilds the line in interchanged order. The symbol **\2** replaces the matched string enclosed in the second pair of **\(** **\)** delimiters, and the symbol **\1** inserts the matched string enclosed in the first pair of **\(** **\)**.

The right side of the substitution inserts the second matched expression (from **\2**), then inserts the = sign again, followed finally with the first part of the line from **\1**.

This may appear involved, but can be immensely valuable in situations requiring rearrangement of a large number of lines.

The next special characters for patterns that we will consider are the bracket characters **[** and **]**. These are used to define the character class. Inside the brackets, put a list of characters that you consider alternatives for the match at that position in the string, and **ed** will match if any one of them appears. For example, to print a line that contains any odd digit, say:

```
g/[13579]/p
```

For even more power and flexibility, you can combine character classes with the star. Find and print all lines that contain a negative number followed by a period. Note that the number may not contain commas:

```
g/-[0123456789]*\./p
```

This will match lines containing the following example strings:

```
-1.
-666.
-3.7.77
```

You can also match all lower case letters by listing them in brackets also, but an abbreviation mechanism simplifies this:

```
g/[a-z]/p
```

will do the job. This can be used for the negative number example above by:

```
g/-[0-9]*\./p
```

Most special characters lose their original meaning within the brackets, but one of the special characters, caret ^, gets a new meaning. If you want to match all but a class of characters, the caret when used immediately after the left bracket will do the job. To match a string that begins with **K** and continues with any character except a number, use

```
/K[^0-9]/
```

which will match

```
KQ
KK
KK9
```

but none of the following:

```
K7
kK0
```

Other special characters may be part of a character class, and will lose their special meaning. However, if you want to match the right bracket, it must appear first in the list. So, to find all occurrences of special characters in the file, type:

```
g/[]^\.*[&]/p
```

Listing funny lines

The **p** command prints lines with graphic characters in them. It will also print lines with non-graphic characters, but these will not show on the screen. For example, a line containing the bell character <**ctrl-G**> will sound your terminal's bell or buzzer, but you will not be able to tell where the bell character occurs within the line.

The **l** command will behave like the **p** command, but if there are non-graphic characters in the line, they will be decoded and printed in octal preceded by a backslash. If a line containing the word **bell** followed by a bell character were printed with **l**, the result would appear

```
bell\007
```

Also, a backspace character <**ctrl-H**> is displayed as the character – overstruck with a <, which will appear simply as < on a CRT. Tab characters are displayed as a – overstruck with a >, which will appear as > on a CRT. If the line being listed with **l** is too long for a screen line, it is separated into two lines, with the backslash character placed at the end of the first line to indicate the split.

All other features of the **p** command apply to the **l** command.

Keeping track of current line

The most commonly used abbreviation in **ed** is the dot, or period, standing for the number of the current line. Many commands have the potential for changing the value of the dot, and it is useful to you to be able to anticipate this change when using the abbreviation.

The influence of each command on the value of the dot can be separated by classes of commands. In general, however, the simple explanation is often true, and is always a good starting point. The current line is the last line to be processed by the previous command. We will use this definition as a first approximation, then refine in the case of each command.

An example of the current line being changed is the substitute command. In the example

```
1,$s/flow/change/
p
```

the current line after the substitutions will be the last line that was modified, and that will be the line that **p** will print.

The **w** command is an exception. The current line is never changed, regardless of any line range selection or how these ranges are developed.

After execution of the **r** command, the current line is the last of the lines read in from the file.

The **d** command sets the current line to the line following the last line deleted, unless the last line in the file was deleted, in which case the new last line becomes the current line.

The line insertion commands **i**, **c**, and **a** all leave the current line as the last line added. If no commands are added, however, the behavior differs. For **i** and **c**, the last line is effectively backed up one, whereas for **a**, it stays the same.

When current line is changed

The time of changing the current line is of importance, also. Normally the current line is not changed until the command is completed.

To illustrate, create a file **semi** by typing:

```
ed
a
begin
second
first
in between
second
last
.
w semi
q
```

Now, edit the file and type all lines from **first** to **second**:

```
ed semi
/first/,/second/p
Q
```

This will cause an error! The reason is that the search command begins with current line set to **$**, so "first" is found on line 3. But the search for "second" also begins with the current line set at **$**, and finds "second" on line two. Thus, the command translates to

```
3,2p
```

which is clearly invalid.

To do what was intended, use the semicolon ; in place of the comma separating the two searches. This forces the current line to be changed after the search for **first** rather than after the entire command is completed. The commands

```
ed semi
/first/;/second/p
Q
```

are not in error and will do what is intended. The result will be

```
first
in between
second
```

The search for **first** still begins with the current line set at **$**. However, after **first** is found, the current line is set to 3, and the search for **second begins** accordingly, and succeeds on line 5.

Finally, to be sure of where the current line is, you can use the **p** command to show you the line. Or, you can have **ed** tell you the number of the current line by typing

```
=
```

To give you a perspective on where you are with respect to the end of the file, type

```
=
```

and **ed** will tell you the number of the last line in the file.

You can put any line number expression before = and it will type the result. For example

```
/next/=
```

will type the number of the next line containing "next" (if there is one). This command = will never change the line number.

More about global commands

All the global commands discussed thus far have been followed by single commands—substitute, print, and delete. You can put several commands after a global command, however, and have each of those commands executed for lines that match.

To change all occurrences of the word **cacophonous** to the word **noisy** and print the three lines that follow, issue the command

```
g/cacophonous/s//noisy/\
.+1,.+3p
```

in which the additional commands are separated by the backslash before the **<RETURN>**. Several commands can be added, and all but the last need the backslash at the end.

This will work for the line-adding commands, as well. To insert a spelling warning before each line that contains the word **occurrance**, issue the command

```
g/occurrance/i\
((the following line needs spelling check))\
.
```

Note that the last line of the **i** group can be entered without a backslash, in which case the line containing only the period must be omitted. This has the same effect as

```
g/occurrance/i\
((the following line needs spelling check))
```

You should not depend upon the setting of the current line in any multiline global command. There are two reasons for this. First, if one of the commands is a substitute, and the string is not found in the matched line, the current line will not be changed.

Secondly, the global command operates in two phases. The first part scans the file for lines that match the string argument. These lines are marked internally in a manner similar to the **k** command

by **ed**. The second phase then executes the commands on each of the marked lines. Therefore, you cannot count on a string search following the **g** to set the current line number.

Again, the **v** command behaves in the same way, except lines that do *not* match the pattern are selected.

Caution is advised when using remembered search arguments, for a similar reason. A search argument is remembered only if the search has been executed. Thus, in a command of the form

```
g/backup/s//reverse/\
s/backin /backing/
```

the first remembered search may use **backup** on some occasion, and "**backin** " on others. The reason for this is that the second phase of the **g** command begins with a remembered search argument of **backup.** After the second line of the multiline command executes, the remembered search argument will be "**backin** ". This will remain throughout the remainder of the second **g** phase.

Thus, it is recommended that you avoid remembered search arguments when using multiline global commands.

Issuing COHERENT commands within ed

While you are using **ed**, you can issue COHERENT commands by prefixing them with the **!** character.

This can be very useful if you need to determine a file name while in the middle of an edit, or if someone has sent you a message, and you want to reply without leaving **ed**.

Thus, to list your directory while in **ed**, type:

```
!lc
```

and **ed** will send the command to COHERENT, and echo a **!** character when the command is finished.

There is no limitation on the type of command that you may issue with this feature. It is even plausible that you want to start another **ed**.

Summary

This section discusses the advanced features of **ed**.

The **ed** command **e** permits you to begin editing a different file without leaving **ed**.

The **r** command reads in lines from another file, while the **w** command writes out the entire file as changed, or selected lines from the file.

The **f** command prints the remembered file name, and can also change it.

Patterns and their uses are described in detail. The special characters

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

aid in pattern construction.

The character **&** when used in a substitute command causes the matched string to be substituted, even if it were a complex pattern. Also, the related symbols **\1** and **\2** replace parts of a matched pattern delimited by **\(** and **\)**.

To help shorten your typing, **ed** provides a remembered search facility.

If some lines of your file have non-graphic characters in them, the **l** command will list the octal values of the non-graphic characters.

Since nearly every **ed** command will default to the *current line* to increase your convenience, this section goes on to discuss how to keep track of the current line, and how various commands change it.

Then expanded use of global commands is shown. You can enter several commands after a global command rather than just one.

You can issue COHERENT commands without leaving **ed**, for example, to list your directory.

6. Command summary

The **ed** commands are summarized in this section.

Line locators, sometimes also called line numbers, are shown enclosed in brackets. The part

[*n*]

indicates an optional line locator, which defaults to the current line if left off except as noted below.

[*n*[,*m*]]

The outer brackets indicate that the range is optional. The inner show that the ending value of the range, *m*, is also optional if the beginning value of the range, *n*, is given. In such a case, the range is *n*,*m*. If the entire range is left off, the range defaults to the current line. The letters *n*, *m*, and *d* will be used within braces to signal line locators.

Notice that string search commands // and ?? may be used wherever line numbers appear.

Other letters appearing within braces are optional parts of commands and are described with the command.

The comma in a line number range can be replaced by a semicolon. See the section on expert editing for details.

Line specifiers

In addition to being simple integers, the *n* and *m* of line ranges can take on symbolic forms. These forms are

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

. (dot) Current line.

$ Last line.

+, - Simple arithmetic with line numbers.

[*n*[,*m*]]*/pattern1/*
String search to match pattern *pattern1* within selected range. Result of search is line number. If no range specified, begin with line following current line and end with current line number after wraparound.

[*n*[,*m*]]?*pattern1*?
Similar to string search with /, but search in reverse direction.

'*m* Number of line with mark *m*.

* Eqivalent to **1,$**.

Commands

. (dot) Print the current line. Also end of **a**ppend, **i**nsert, or **c**hange.

[*n*]= Type the given line number. If *n* is omitted, type the number of the last line.

[*n*]**&** Print a screen of lines. Equivalent to *n*,*n*+22**p** unless *n* is near the end of the file, in which case it is equivalent to *n*,**$p**.

! *line* Pass the given *line* to the shell **sh** for execution.

? Print a brief description of the most recent error.

[*n*]**a** Append lines to file after line *n*. Terminate added text with a line containing only a dot.

[*n*[,*m*]]**c**
Replace specified lines with lines that follow. Text ended by a line containing only dot.

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

e [*file*] Edit a new file *file*. Gives error if there are unsaved changes. If error is given, reissue the **e** *file*, and then **ed** will exit.

E [*file*]
Edit a new file *file*. Do not give error if there are unsaved changes.

f [*file*] Set the remembered file name to *file*. If [*file*] is not specified, type the currently remembered file name.

[*n*[,*m*]]**g**/[*pattern*]/*commands*
Globally execute *commands* for each line in the specified range. If no range is specified, all lines are searched. Search for lines containing the *pattern* and internally mark

them. Then, for lines so marked, execute *commands*. The commands (except for substitute) may extend over several lines, with all but the last terminated by '\'.

[*n*]**i** Insert lines before line *n*. Terminate new text with line containing only '.' dot.

[*n*[,*m*]]**j**[**p**]
Join all lines in specified range into one line. If *m* is not specified, use range *n*,*n*+1. With optional **p**, print resulting line.

[*n*]**k***x* Mark line *n* with marker *x* (lower-case letter). This command does not change the current line.

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

[*n*[,*m*]]**m**[*d*]
Move selected lines of text to after line *d*.

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

[*n*[,*m*]][**p**]
Print selected lines on terminal. **p** can be omitted.

q Exit editor. Gives error if there are unsaved changes. If error is given, issue another **q**, and then **ed** will exit.

Q Exit editor; give no error if unsaved changes.

[*n*]**r** [*file*]
Read file *file* into current file following specified line, or after last line in memory if no line selected.

[*n*[,*m*]]**s**[*k*]/*pattern1*/*pattern*2/[**g**][**p**]
Within selected line number range, search for *k*th pattern *pattern1* and, if found, replace with *pattern2*. If *k* is not specified, search for the first. If **g** follows, replace all *pattern1* found within each line in range. If **p** follows, print current line when done. If *pattern1* not specified, use remembered search argument instead.

[*n*[,*m*]]**t**[**d**]
: Copy selected lines to the point before destination line **d**, which defaults to the current line.

[*n*]**u**[**p**]
: Undo effect of last substitute command. If optional **p** specified, print undone line.

[*n*[,*m*]]**v**/[*pattern*]/*commands*
: Globally execute *commands* for each line in the specified range. If no range is specified, all lines are searched. Search for lines that do *not* contain the *pattern* and internally mark them. Then, for lines so marked, execute *commands*. The commands (except for substitute) may extend over several lines, with all but the last terminated by '\'.

[*n*[,*m*]]**w** [*file*]
: Write selected lines to file *file* (defaults to current file). Line selection defaults to **1,$** if *n* and *m* are not specified.

[*n*[,*m*]]**W** [*file*]
: As in **w** command, except append lines to lines already existing in file *file*.

wq *file*
: Write all lines to the current file and quit the editor.

x
: Encrypt/decrypt text using the system library routine **crypt**. **ed** prompts for an encryption password and applies the resulting key to encrypt or decrypt on each subsequent **e**, **r**, or **w** command. An empty password turns off encryption.

Pattern elements

String searches and substitute commands employ special characters used to describe patterns. The following characters appear in *pattern1* of the substitute command, and between / or ? of the search commands.

Non-special characters within the pattern match themselves. Special characters that make up patterns are:

^
: Matches beginning of line, unless it appears immediately after '[', in which case it means "match any character but the following".

$ Matches end of line.

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

. (dot) Matches any character except newline.

[*chars***]**
Matches any of the following **chars** up to **]**. Ranges of letters or digits may be indicated using "**–**".

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

**** Disregard special meaning of following character.

\(Beginning delimiter to define substring of *pattern1*; ending delimiter is **\)**. Used in conjunction with *****n*, below.

pattern2 or the replacement part of the substitute command uses the following special characters.

& Insert characters that were matched by **pattern1**.

\1, \2, ...
Replace part of matched string delimited by **n**th occurrence of delimiters **\(** and **\)**.

Options

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

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

m Allow multiple commands per line.

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

p Prompt with '*' for each command.

s Match lower case letters in a *pattern* to both upper case and lower case text characters.

v Print verbose versions of error messages.

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

The '−' command line option resets the **c** option. The '−x' command line option causes **ed** to encrypt and decrypt text written to and read from files, as with the **x** command.

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

```
ED = + cv
```

options may be set with the '+' prefix or reset with the '−' prefix.

Index

User Reaction Report

To keep this manual and COHERENT free of bugs and facilitate future improvements, we would appreciate receiving your reactions. Please fill in the appropriate sections below and mail to us. Thank you.

Mark Williams Company
1430 W. Wrightwood Avenue
Chicago, IL 60614

Name: _____

Company: _____

Address: _____

_____

Phone: _____ Date: _____

Version and hardware used: _____

Did you find any errors in the manual? _____

_____

_____

Can you suggest any improvements to the manual? _____

_____

_____

Did you find any bugs in the software? _____

_____

_____

Can you suggest improvements or enhancements to the software?

_____

_____

_____

Additional comments: (Please use other side.)