

DOMAIN/IX Text Processing Guide

Order No. 005802
Revision 00
Software Release 9.0

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

Apollo Computer Inc. reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Apollo Computer Inc. to determine whether any such changes have been made.

THE TERMS AND CONDITIONS GOVERNING THE SALE OF APOLLO COMPUTER INC. HARDWARE PRODUCTS AND THE LICENSING OF APOLLO COMPUTER INC. SOFTWARE CONSIST SOLELY OF THOSE SET FORTH IN THE WRITTEN CONTRACTS BETWEEN APOLLO COMPUTER INC. AND ITS CUSTOMERS. NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO STATEMENTS REGARDING CAPACITY, RESPONSE-TIME PERFORMANCE, SUITABILITY FOR USE OR PERFORMANCE OF PRODUCTS DESCRIBED HEREIN SHALL BE DEEMED TO BE A WARRANTY BY APOLLO COMPUTER INC. FOR ANY PURPOSE, OR GIVE RISE TO ANY LIABILITY BY APOLLO COMPUTER INC. WHATSOEVER.

IN NO EVENT SHALL APOLLO COMPUTER INC. BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATING TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF APOLLO COMPUTER INC. HAS BEEN ADVISED, KNEW, OR SHOULD HAVE KNOWN OF THE POSSIBILITY OF SUCH DAMAGES.

THE SOFTWARE PROGRAMS DESCRIBED IN THIS DOCUMENT ARE CONFIDENTIAL INFORMATION AND PROPRIETARY PRODUCTS OF APOLLO COMPUTER INC. OR ITS LICENSORS.

THIS SOFTWARE AND DOCUMENTATION ARE BASED IN PART ON THE FOURTH BERKELEY SOFTWARE DISTRIBUTION UNDER LICENSE FROM THE REGENTS OF THE UNIVERSITY OF CALIFORNIA.

© 1985 Apollo Computer Inc. All rights reserved.

Printed in U.S.A.

First Printing: July 1985

This document was formatted using the **troff** text formatter distributed with DOMAIN®/IX™ software.

APOLLO and DOMAIN are registered trademarks of Apollo Computer Inc. AEGIS, DGR, DOMAIN/IX, DPSS, DSEE, D3M, GMR, and GPR are trademarks of Apollo Computer Inc.

PREFACE

The DOMAIN[®]/IX[™] *Text Processing Guide* and its companion volume, *The DOMAIN/IX User's Guide* consist of those papers normally included in Volumes 2A, 2B, and 2C of the UNIX[†] *Programmer's Manual* as supplied by Bell Telephone Labs and the University of California at Berkeley. The papers in these books have been revised where necessary to reflect the DOMAIN system environment. However, we have tried to remain aware of the history of UNIX as a multiuser system, and have included the more important references to operations conducted at terminals.

Audience

This *Text Processing Guide* is intended for users who are familiar with UNIX software, AEGIS[™] software, and DOMAIN networks. We recommend that you read one of the following tutorial introductions if you are not already familiar with UNIX.

- Bourne, Stephen W. *The UNIX System*. Reading: Addison-Wesley, 1982.
- Kernighan, Brian W. and Rob Pike. *The UNIX Programming Environment*, Englewood Cliffs, Prentice-Hall, 1984.
- Thomas, Rebecca and Jean Yates. *A User Guide to the UNIX System*. Berkeley: Osborne/McGraw-Hill, 1982.

This document also assumes a basic familiarity with the DOMAIN system. The best introduction to AEGIS and the DOMAIN system is *Getting Started With Your DOMAIN System* (Order No. 002348). This manual explains how to use the keyboard and display, read and edit text, and create and execute programs. It also shows how to request DOMAIN system services using interactive commands.

The Structure of This Document

This guide is divided into two sections and an appendix.

Section 1 deals with the text editors **ed**, **ex**, and **vi**. It also provides a brief introduction to the DOMAIN system's DM editor.

Section 2 covers the formatters **troff** and **nroff**, the macro packages **-me**, **-ms**, and **-mm**, and the preprocessors **eqn** and **tbl**.

Appendices The Appendices are all UNIX papers related to text processing. They are presented here in their original form.

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

Related Volumes

The *DOMAIN/IX User's Guide* (Order No. 005802) is the first volume you should read. It explains how DOMAIN/IX works, and contains extensive material on the Bourne Shell, C Shell, and the communications utilities **Mail** and **uucp**.

The *DOMAIN/IX Command Reference for System V* (Order No. 005798) describes all the UNIX System V shell commands supported by the *sys5* version of DOMAIN/IX.

The *DOMAIN/IX Programmer's Reference for System V* (Order No. 005799) describes all the UNIX System V system calls and library functions supported by the *sys5* version of DOMAIN/IX.

The *DOMAIN/IX Command Reference for BSD4.2* (Order No. 005800) describes all the BSD4.2 UNIX shell commands supported by the *bsd4.2* version of DOMAIN/IX.

The *DOMAIN/IX Programmer's Reference for System V* (Order No. 005801) describes all the BSD4.2 UNIX system calls and library functions supported by the *bsd4.2* version of DOMAIN/IX.

The *DOMAIN C Language Reference* (Order No. 002093) describes C program development on the DOMAIN system. It lists the features of C, describes the C library, and gives information about compiling, binding, and executing C programs.

The *DOMAIN System Command Reference* (Order No. 002547) gives information about using the DOMAIN system and describes the DOMAIN commands.

The two-volume *DOMAIN System Call Reference* (Volume I Order No. 007196, Volume II Order No. 007194) describes calls to operating system components that are accessible to user programs.

Documentation Conventions

Unless otherwise noted in the text, this manual uses the following symbolic conventions.

- command** Command names and command-line options are set in bold type. These are commands, letters, or symbols that you must use literally.
- output Output returned by programs or commands is shown in Roman type.
- [optional] Square brackets enclose optional items in formats and command descriptions.
- ... Horizontal ellipses indicate that the preceding item can be repeated one or more times.

Preface

- name**[*x*] Single numbers or numbers and letters enclosed in brackets immediately following the name of a UNIX command or library function refer to the section where you can find reference information on the command or function in the *DOMAIN/IX Command Reference* or the *DOMAIN/IX Programmer's Reference*.
- ↑*x* A control character, where *x* is the character.
- SMALL CAPS** We use small caps for acronyms and key names; e.g., ASCII and **RETURN**. Note that in tutorial material, we place a box around the name of a key.
- filename* We use italics to represent generic, or meta- names in example command lines, and also to represent a character that stands for another character, as in **dx** where *x* is a digit. In text, the names of files written or read by programs are set in italics.

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. In order to make it easy for you to communicate with us, we provide the User Change Request (UCR) system for software-related comments, and the Reader's Response form for documentation comments. By using these formal channels, you make it easy for us to respond to your comments.

You can get more information about how to submit a UCR by consulting the *DOMAIN System Command Reference*. Refer to the CRUCR (Create User Change Request) command. You can also get more information by typing:

/com/help crucr

in any UNIX or AEGIS shell. There is a Reader's Response form at the back of this manual. We'd appreciate it if you would take the time to fill it out when you're ready to comment on this document.

C

C

C

C

C

CONTENTS

1. An ed Tutorial 1-1
 - 1.1 INTRODUCTION 1-1
 - 1.2 STARTING ED 1-1
 - 1.3 CREATING TEXT [a] 1-2
 - 1.4 ERROR MESSAGES [?] 1-3
 - 1.5 WRITING TEXT TO A FILE [w] 1-3
 - 1.6 LEAVING ed [q] 1-4
 - 1.6.1 Exercise 1 1-4
 - 1.7 READING TEXT FROM A FILE [e] 1-4
 - 1.8 READING TEXT FROM A FILE [r] 1-5
 - 1.8.1 Exercise 2 1-5
 - 1.9 PRINTING THE CONTENTS OF THE BUFFER [p] 1-6
 - 1.9.1 Exercise 3 1-7
 - 1.10 THE CURRENT LINE [.] 1-7
 - 1.11 DELETING LINES [d] 1-8
 - 1.11.1 Exercise 4 1-9
 - 1.12 MODIFYING TEXT [s] 1-9
 - 1.12.1 Exercise 5 1-11
 - 1.13 CONTEXT SEARCHING 1-11
 - 1.13.1 Exercise 6 1-13
 - 1.14 CHANGE [c] AND INSERT [i] 1-14
 - 1.14.1 Exercise 7 1-15
 - 1.15 MOVING TEXT [m] 1-15
 - 1.16 THE GLOBAL COMMANDS [g, v] 1-16
 - 1.17 SPECIAL CHARACTERS 1-17
 - 1.18 SUMMARY OF COMMANDS AND LINE NUMBERS 1-19
2. The ex Reference Manual 2-1
 - 2.1 INTRODUCTION 2-1
 - 2.2 USAGE 2-1
 - 2.3 FILE MANIPULATION 2-2
 - 2.3.1 Current File 2-2
 - 2.3.2 Alternate File 2-2
 - 2.3.3 Filename Expansion 2-2
 - 2.3.4 Multiple Files and Named Buffers 2-3
 - 2.3.5 Read Only 2-3
 - 2.4 EXCEPTIONAL CONDITIONS 2-3
 - 2.4.1 Errors and Interrupts 2-3
 - 2.4.2 Recovering From Hangups and Crashes 2-3
 - 2.5 EDITING MODES 2-4
 - 2.6 COMMAND STRUCTURE 2-4
 - 2.7 COMMAND PARAMETERS 2-4
 - 2.7.1 Command Variants 2-5
 - 2.7.2 Flags After Commands 2-5

- 2.7.3 Comments 2-5
- 2.7.4 Multiple Commands per Line 2-5
- 2.7.5 Reporting Large Changes 2-5
- 2.8 COMMAND ADDRESSING 2-6
 - 2.8.1 Addressing Primitives 2-6
 - 2.8.2 Combining Addressing Primitives 2-6
- 2.9 COMMAND DESCRIPTIONS 2-7
- 2.10 REGULAR EXPRESSIONS 2-18
 - 2.10.1 Regular Expressions 2-18
 - 2.10.2 Magic and Nomagic 2-18
 - 2.10.3 Regular Expression Summary 2-18
 - 2.10.4 Combining Regular Expression Primitives 2-19
 - 2.10.5 Substitute Replacement Patterns 2-19
- 2.11 OPTION DESCRIPTIONS 2-20
- 2.12 LIMITATIONS 2-25
- 3. An Introduction to Display Editing With vi 3-1
 - 3.1 INTRODUCTION 3-1
 - 3.2 GETTING STARTED 3-1
 - 3.2.1 Notational Conventions 3-2
 - 3.2.2 Vi and the VT100 Emulator Program 3-2
 - 3.2.3 Keyboard Mapping 3-2
 - 3.2.4 Specifying Terminal Type 3-3
 - 3.2.5 Editing a File 3-4
 - 3.2.6 The Buffer 3-5
 - 3.2.7 View 3-5
 - 3.2.8 Arrow Keys 3-5
 - 3.2.9 Special Characters: ESC, RETURN and DEL 3-5
 - 3.2.10 Getting Out of the Editor 3-6
 - 3.3 MOVING AROUND IN THE FILE 3-7
 - 3.3.1 Scrolling and Paging 3-7
 - 3.3.2 Searching, Goto, and Previous Context 3-7
 - 3.3.3 Moving Around on the Screen 3-8
 - 3.3.4 Moving Within a Line 3-9
 - 3.3.5 Summary of Cursor Movement and Scrolling 3-10
 - 3.4 MAKING SIMPLE CHANGES 3-10
 - 3.4.1 Inserting 3-10
 - 3.4.2 Making Small Corrections 3-11
 - 3.4.3 More Corrections: Operators 3-12
 - 3.4.4 Operating on Lines 3-12
 - 3.4.5 Undo 3-13
 - 3.4.6 Summary of Insert/Delete Functions 3-13
 - 3.5 MOVING, REARRANGING, AND DUPLICATING TEXT 3-13
 - 3.5.1 Low Level Character Motions 3-13
 - 3.5.2 Higher-Level Text Objects 3-14
 - 3.5.3 Rearranging and Duplicating Text 3-15

- 3.5.4 Summary of Higher-Level Motions and Objects 3-17
- 3.6 HIGH LEVEL COMMANDS 3-17
 - 3.6.1 Writing, Quitting, Editing New Files 3-17
 - 3.6.2 Escaping to a Shell 3-17
 - 3.6.3 Marking and Returning 3-18
 - 3.6.4 Adjusting the Screen 3-18
- 3.7 ADVANCED TOPICS 3-18
 - 3.7.1 Editing on Slow Terminals 3-19
 - 3.7.2 Options, Set, and Editor Startup Files 3-20
 - 3.7.3 Recovering Lost Lines 3-21
 - 3.7.4 Recovering Lost Files 3-21
 - 3.7.5 Continuous Text Input 3-22
 - 3.7.6 Features for Program Editing 3-22
 - 3.7.7 Filtering Portions of the Buffer 3-23
 - 3.7.8 Commands for Editing LISP 3-23
 - 3.7.9 Macros 3-24
- 3.8 ABBREVIATIONS 3-25
 - 3.8.1 Word Abbreviations 3-25
 - 3.8.2 Editor Command Abbreviations 3-25
- 3.9 MORE DETAILS 3-25
 - 3.9.1 Line Representation in the Display 3-26
 - 3.9.2 Counts 3-26
 - 3.9.3 More File Manipulation Commands 3-27
 - 3.9.4 More About Searching for Strings 3-28
 - 3.9.5 More About Input Mode 3-29
 - 3.9.6 Uppercase Only Terminals 3-30
 - 3.9.7 Vi and ex 3-30
 - 3.9.8 Open Mode: vi on Hardcopy Terminals and "Glass TTY's" 3-31
- 3.10 A SUMMARY OF VI COMMANDS 3-31
 - 3.10.1 Entry and Exit 3-32
 - 3.10.2 Cursor and Page Motion 3-33
 - 3.10.3 Searches 3-36
 - 3.10.4 Text Insertion 3-37
 - 3.10.5 Text Deletion 3-37
 - 3.10.6 Text Replacement 3-38
 - 3.10.7 Moving Text 3-38
 - 3.10.8 Miscellaneous Commands 3-40
 - 3.10.9 Special Insert Characters 3-41
 - 3.10.10 ":" Commands 3-42
 - 3.10.11 Special Arrangements for Startup 3-43
 - 3.10.12 Set Commands 3-43
- 4. An Introduction to the DM Editor 4-1
 - 4.1 THE DISPLAY MANAGER EDITOR 4-1
 - 4.2 OPENING AN EDIT PAD 4-2
 - 4.3 SAVING THE CONTENTS OF AN EDIT PAD 4-3
 - 4.4 EDIT PAD MODES 4-3

- 4.5 INSERTING CHARACTERS 4-4
 - 4.5.1 Inserting a Text String 4-4
 - 4.5.2 Inserting an End-of-File Mark 4-5
 - 4.5.3 Inserting a TAB 4-5
- 4.6 DELETING TEXT 4-5
 - 4.6.1 Deleting Characters 4-5
 - 4.6.2 Deleting Words 4-5
 - 4.6.3 Deleting Lines 4-6
- 4.7 DEFINING A RANGE OF TEXT 4-6
- 4.8 COPYING, CUTTING, AND PASTING TEXT 4-6
 - 4.8.1 Using Paste Buffers 4-6
 - 4.8.2 Copying Text 4-7
 - 4.8.3 Cutting Text 4-7
 - 4.8.4 Pasting Text 4-8
- 4.9 USING REGULAR EXPRESSIONS 4-8
- 4.10 SEARCHING FOR TEXT 4-8
 - 4.10.1 Case Sensitivity 4-9
 - 4.10.2 Cancelling a Search Operation 4-9
- 4.11 SUBSTITUTING TEXT 4-10
 - 4.11.1 Substituting All Occurrences of a String 4-10
 - 4.11.2 Substituting the First Occurrence of a String 4-10
 - 4.11.3 Changing the Case of Letters 4-10
- 4.12 UNDOING PREVIOUS COMMANDS 4-10

Chapter 1: An ed Tutorial

1.1 INTRODUCTION

Ed is a line-oriented text editor that supports a wide variety of terminals (including all DOMAIN nodes running DOMAIN/IX. It allows the interactive creation and modification of text based on your directions. The text may be a document, a program, or perhaps data for a program.

This introduction is meant to simplify learning **ed**. The recommended way to learn **ed** is to read this document, and simultaneously use **ed** to follow the examples. Then, read the description of **ed** in the *DOMAIN/IX Command Reference*.

As you read this chapter, we recommend that you also do the exercises. They cover material not completely discussed in the actual text. There is a summary of **ed** commands at the end of this chapter.

Since this chapter is an introduction and a tutorial, no attempt is made to cover more than a part of the facilities that **ed** offers (although this fraction includes the most frequently used parts). Since there is not enough space here to explain basic UNIX procedures, we will assume that you know how to log in to a UNIX shell, and that you have a general understanding of a *file*.

1.2 STARTING ED

Once you log in, you may invoke **ed** in any shell window by typing

```
ed  
[RETURN]
```

You are now ready to go — **ed** is waiting for your instructions.

Note: You may invoke **ed** in either a Bourne Shell or a C Shell using the procedure shown above. Also, you may invoke **ed** in an AEGIS Shell by either typing

```
$ /bin/ed  
[RETURN]
```

or setting the AEGIS Shell's command search rules to include */bin*, then typing

```
ed  
[RETURN]
```

as shown in the first example above.

1.3 CREATING TEXT [a]

Suppose you want to create some text starting from scratch. Perhaps you are typing the very first draft of a paper; clearly, it will have to start somewhere, and undergo modifications later. This section will show how to create some text, just to get started. Later we'll talk about how to change text.

When you first start **ed**, it is like working with a blank piece of paper – there is no text or information present. You must supply the text. Usually, this is done by typing in the text, or by reading it into **ed** from a file. In this example, we will type in some text. Later, we will return to learn how to read files.

First a bit of terminology. In **ed** jargon, the text being worked on is said to be “kept in a buffer.” Think of the buffer as a work space, if you like, or simply as the information that you are going to be editing. In effect the buffer is like the piece of paper, on which we will write things, then change some of them, and finally file the whole thing away for another day.

You tell **ed** what to do to your text by typing instructions called “commands.” Most commands consist of a single letter, which must be typed in lowercase. Each **ed** command is typed on a separate line. (Sometimes the command is preceded by information about what line or lines of text are to be affected. We will discuss this shortly.) **Ed** makes no response to most commands; there is no prompting or typing of messages like “ready”. (This silence is preferred by experienced users, but sometimes presents a problem for beginners.)

The first command is *append*, written as the letter

a

all by itself. It means “append (or add) text lines to the buffer, as I type them in.” Appending is rather like writing fresh material on a piece of paper.

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

a

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

The only way to stop appending is to type a line that contains only a period. The “.” is used to tell **ed** that you have finished appending. If **ed** seems to be ignoring you, type an extra line with just “.” on it. You may find you've added some extra lines to your text, which you'll have to take out later.

When the append command is done, the buffer will contain the three lines

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

The “**a**” and “.” aren’t there, because they are not text.

To add more text, just issue another **a** command, and continue typing.

1.4 ERROR MESSAGES [?]

If at any time you make an error in the commands you type to **ed**, it will tell you by displaying

```
?
```

The editor’s response does not explain your error; make certain you are entering a valid command.

1.5 WRITING TEXT TO A FILE [w]

It’s likely that you’ll want to save your text for later use. To write out the contents of the buffer onto a file, use the *write* command

```
w
```

followed by the filename you want to write on. This will copy the buffer’s contents onto the specified file (destroying any previous information on the file). To save the text on a file named **practice**, for example, type

```
w practice
```

Leave a space between **w** and the file name. **Ed** responds by printing the number of characters it wrote out. In this case, **ed** would respond with

```
68
```

(Remember that blanks and the RETURN character at the end of each line are included in the character count.) Writing a file just makes a copy of the text; the buffer’s contents are not disturbed, so you can go on adding lines to it. This is an important point. **Ed** always works on a copy of a file, rather than on the file itself. No change in the contents of a file takes place until you give a **w** command.

Note: Writing out the text onto a file from time to time as it is being created is a good idea. That way, if the system crashes or if you make some horrible mistake, you will only lose the text currently in the buffer. Text that has been written to a file is relatively safe.

1.6 LEAVING ed [q]

To terminate an **ed** session, save your text by writing it onto a file using the **w** command, and then type the command

q

which stands for **quit**. The system will respond by returning control to the Shell, which will display its prompt character. At this point your buffer vanishes, with all its text, which is why you want to write it out before quitting.

Note: Actually, **ed** will print **?** if you try to quit without writing. At that point, write if you want; if not, typing another **q** will get you out.

1.6.1 Exercise 1

Enter **ed** and create some text using

a
lines of text
lines of text
lines of text
 .

Write it out using **w**. Then leave **ed** by giving it the **q** command. After you return to the Shell, print the file, to see the results.

1.7 READING TEXT FROM A FILE [e]

A common way to get text into the buffer is to read it from a file in the file system. This is what you do to edit text that you saved with the **w** command in a previous session. The **edit** command **e** puts the entire contents of a file into the buffer. So if you had saved the three lines "Now is the time", etc., with a **w** command in an earlier session, the **ed** command

e practice

would fetch the entire contents of the file "practice" into the buffer, and respond

68

— the number of characters in "practice."

Note: If anything is already in the buffer when you do an **e**, it will be overwritten (deleted).

If you use the **e** command to read a file into the buffer, then you need not use a file name after a subsequent **w** command; **ed** remembers the last file name used in an **e** command, and **w** will write on this file. Thus

a good way to operate is

```
ed
e file
  editing session
w
q
```

This way, you can simply say **w** from time to time and be secure in the knowledge that as long as you used the correct file name with **e**, you are writing into the proper file each time.

You can find out at any time what file name **ed** is remembering by typing the *file* command **f**. In this example, if you typed

```
f
```

ed would reply

```
practice
```

1.8 READING TEXT FROM A FILE [r]

Sometimes you want to read a file into the buffer without destroying anything that is already there. This is done by the *read* command **r**. The command

```
r practice
```

will read the file "practice" into the buffer; it adds it to the end of whatever is already in the buffer. If you do a read after an edit:

```
e practice
r practice
```

the buffer will contain *two* copies of the text (six lines).

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

Like the **w** and **e** commands, **r** prints the number of characters read in, after the reading operation is complete.

Generally speaking, **r** is used less than **e**.

1.8.1 Exercise 2

Experiment with the **e** command – try reading and printing various files. You may get an error **?name**, where **name** is the name of a file; this means that the file doesn't exist, typically because you spelled the file name wrong, or perhaps because you are not allowed to read or write on it. Try alternately reading and appending to see that they work

similarly. Verify that

ed *filename*

is exactly equivalent to

ed
e *filename*

What does

f *filename*

do?

1.9 PRINTING THE CONTENTS OF THE BUFFER [p]

To *print* or list the contents of the buffer (or parts of it) on the terminal, use the print command

p

To do this, specify the lines where you want printing to begin and where you want it to end, separated by a comma, and followed by the letter **p**. Thus to print the first two lines of the buffer, for example, (that is, lines 1 through 2) say

1,2p

(Starting line=1, ending line=2, print.) **Ed** will respond with

Now is the time
for all good men

If you wanted to print all the lines in the buffer, you could use **1,3p** as above if you knew there were exactly 3 lines in the buffer. But in general, you don't know how many there are, so what do you use for the ending line number? **Ed** provides a shorthand symbol for "line number of last line in buffer" – the dollar sign **\$**. Use it this way:

1,\$p

This will print all the lines in the buffer (line 1 to last line.) If you want to stop the printing before it is finished, type ↑I. This sends **ed** an interrupt, causing it to display its

?

prompt and wait for the next command.

To print the last line of the buffer, you could use

,\$p

but **ed** lets you abbreviate this to

\$p

You can print any single line by typing the line number followed by a **p**.

Thus

1p

produces the response

Now is the time

which is the first line of the buffer.

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

\$

ed will print the last line of the buffer.

You can also use **\$** in combinations like

\$-1,\$p

which prints the last two lines of the buffer.

1.9.1 Exercise 3

As before, create some text using the **a** command, then experiment with the **p** command. You will find, for example, that you can't print line 0 or a line beyond the end of the buffer, and that attempts to print a buffer in reverse order by saying

3,1p

don't work.

1.10 THE CURRENT LINE [.]

Suppose your buffer still contains the six lines that you have just typed

1,3p

and **ed** has printed the three lines for you. Try typing just

p

with no line numbers. This will print

to come to the aid of their party.

which is the third line of the buffer. In fact, it is the last (most recent) line that you have done anything with. You can repeat this **p** command without line numbers, and it will continue to print line 3.

This is because **ed** maintains a record of the last line you did anything to (in this case, line 3, which you just printed). This most recent line is referred to by the shorthand symbol

.

(pronounced "dot"). Dot is a line number in the same way that **\$** is; it

means exactly “the current line,” or loosely, “the line you most recently did something to.” You can use it in several ways. One possibility is to say

.,\$p

This will print all the lines from (including) the current line to the end of the buffer. In our example, these are lines 3 through 6.

Some commands change the value of dot, while others do not. The **p** command sets dot to the number of the last line printed; the last command will set both **.** and **\$** to 6.

Dot is most useful when used in combinations like this one:

+.1

(or equivalently, **+.1p**). This means “print the next line” and is a handy way to step slowly through a buffer. You can also say

-.1 (or -.1p)

which means “print the line before the current line.” This enables you to go backwards if you wish. Another useful combination is something like

-.3, -.1p

which prints the previous three lines.

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

.=

Ed will respond by printing the value of dot.

Let’s summarize some things about the **p** command and dot. Essentially, **p** can be preceded by 0, 1, or 2 line numbers. If there is no line number given, it prints the “current line,” the line that dot refers to. If there is one line number given (with or without the letter **p**), it prints that line and sets dot to it; and if there are two line numbers, it prints all the lines in that range (and sets dot to the last line printed.) If two line numbers are specified, the first can’t be bigger than the second. (See Exercise 2.)

Typing a single return will cause printing of the next line – it’s equivalent to **+.1p**. Try it. Try typing a **-**; you will find that it’s equivalent to **-.1p**.

1.11 DELETING LINES [d]

Suppose you want to get rid of the three extra lines in the buffer. This is done by the *delete* command **d**. The **d** command deletes lines instead of printing them, but is otherwise similar to **p**. The lines to be deleted are specified for **d** exactly as they are for **p**:

starting line, ending line d

Thus, the command

4,\$d

deletes lines 4 through the end. Now there are three lines left, as you can check by using

1,\$p

Notice that \$ now is line 3. Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to \$.

1.11.1 Exercise 4

Experiment with **a**, **e**, **r**, **w**, **p** and **d** until you are sure that you know what they do, and until you understand how dot, \$, and line numbers are used.

Try using line numbers with **a**, **r** and **w** as well. You will find that **a** will append lines *after* the line number that you specify (rather than after dot); that **r** reads a file into the buffer *after* the line number you specify (not necessarily at the end of the buffer); and that **w** will write out exactly the lines you specify, not necessarily the whole buffer. These variations are sometimes handy. For instance, you can insert a file at the beginning of a buffer by saying

Or *filename*

and you can enter lines at the beginning of the buffer by saying

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

Notice that **.w** is *very* different from

```
.
w
```

1.12 MODIFYING TEXT [s]

The **s** (substitute) command is used to change individual words or letters within a line or group of lines. You can use it to correct spelling mistakes and typing errors.

Suppose that due to a typing error, line 1 says

Now is th time

— the “e” has been omitted from “the.” You can use **s** to fix this, as shown below

1s/th/the/

This says: "in line 1, substitute for the characters th the characters the ." To verify that it works, use the

p

command. **Ed** will print the line, which should now read

Now is the time

Notice that **ed** set "dot" to the line where the substitution took place. We know this because the **p** command printed that line. Dot is always set this way when you use the **s** command.

The general way to use the substitute command is

starting_line#, ending_line# s/change this/to this/

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in all the lines between *starting_line#* and *ending_line#*. Only the first occurrence on each line is changed, however. If you want to change every occurrence, see Exercise 5. The rules for line numbers are the same as those for **p**, except that dot is set to the last line changed.

Note: If no substitution took place, dot is not changed. This causes an error message (?) to be printed as a warning.

Thus you can say

1,\$s/speling/spelling/

and correct the first spelling mistake on each line in the text. If there were a second (or a third) instance of "speling" on any line, it would not be corrected.

If no line numbers are given, the **s** command assumes you mean "make the substitution on line dot", so it changes things only on the current line. This leads to the very common sequence

s/something/something else/p

which makes some correction on the current line, and then prints it, so you can see if the substitution worked out right. If it didn't, you can try again. (Notice that there is a **p** on the same line as the **s** command. With few exceptions, **p** can follow any command; no other multi-command lines are legal.)

It's also legal to say

s/string//

which means "change *string* to nothing," which is the same as saying "delete *string*." This is useful for deleting extra words in a line or removing extra letters from words. For instance, if you had

Nowxx is the time
you can say

s/xx//p

to get

Now is the time

Notice that // (two adjacent slashes) means “no characters”, not a blank.

1.12.1 Exercise 5

Experiment with the substitute command. See what happens when you substitute for some word on a line with several occurrences of that word. For example, do this:

a
the other side of the coin
.
s/the/on the/p

You will get

on the other side of the coin

A substitute command changes only the first occurrence of the first string. You can change all occurrences by adding a **g** (for “global”) to the **s** command, like this:

s/string1/string2/gp

Try other characters instead of slashes to delimit the two sets of characters in the **s** command — any character other than the blank or the tab should work.

(If you get funny results using any of the characters

^ . \$ [* \ &

read the section on “Special Characters.”)

1.13 CONTEXT SEARCHING

Suppose you have the original three line text in the buffer:

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

And suppose you want to find the line that contains “their” because you want to change it to “the.” Since there are only three lines in the buffer, it’s pretty easy to keep track of what line the word “their” is on. But if the buffer contained several hundred lines, and you’d been making changes, deleting and rearranging lines, and so on, it wouldn’t be easy to

know what text was on a given line. Context searching is simply a method of specifying the desired line by specifying some of the text that's on the line, rather than specifying its line number.

The way to say "search for a line that contains *this particular string of characters*" is to type

/this particular string of characters/

delimited, as above, by slashes. For example, the **ed** command

/their/

is a context search which is sufficient to find the desired line; it will locate the next occurrence of the characters between slashes ("their"). It also sets dot to that line and prints the line for verification:

to come to the aid of their party.

"Next occurrence" means that **ed** starts looking for the string at line **.+1** (dot plus one), searches to the end of the buffer, then continues at line 1 and searches to line dot. (That is, when the search reaches line \$ it "wraps around" to line 1 and continues searching.) It scans all the lines in the buffer until it either finds the desired line or gets back to dot again. If the given string of characters can't be found in any line, **ed** types the error message

?

Otherwise, it prints the first line in which the specified text appears.

You can combine the search for the desired line with the substitution using the syntax below.

/their/s/their/the/p

which will yield

to come to the aid of the party.

There were three parts to that last command.

1. context search for the desired line
2. make the substitution
3. print the line

The expression **/their/** is a context search expression. In essence, all context search expressions are like this — a string of characters delimited by slashes. Context searches are interchangeable with line numbers, so they can be used by themselves to find and print a desired line, or as line numbers for some other command, like **s**. They were used both ways in the examples above.

Suppose the buffer contains the three familiar lines

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

Then the **ed** line numbers

```
/Now/+1
/good/
/party/-1
```

are all context search expressions, and they all refer to the same line (line 2). To make a change in line 2, you could say

```
/Now/+1s/good/bad/
```

or

```
/good/s/good/bad/
```

or

```
/party/-1s/good/bad/
```

The choice is dictated only by convenience. You could print all three lines by typing

```
/Now/,/party/p
```

or

```
/Now/,/Now/+2p
```

or by any number of similar combinations. The first one of these might be better if you don't know how many lines are involved. (Of course, if there were only three lines in the buffer, you'd use

```
1,$p
```

but not if there were several hundred.)

The basic rule is: a context search expression is the same as a line number, so it can be used wherever a line number is needed.

1.13.1 Exercise 6

Experiment with context searching. Try a body of text with several occurrences of the same string of characters, and scan through it using the same context search.

Try using context searches as line numbers for the substitute, print, and delete commands. (They can also be used with **r**, **w**, and **a**.)

Try context searching using **?text?** instead of **/text/**. This scans lines in the buffer in reverse order. This is sometimes useful if you go too far while looking for some string of characters; it's an easy way to back up. (If you get funny results with any of the characters

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

read the section on “Special Characters.”)

Ed provides a shorthand for repeating a context search for the same string. For example, the **ed** line number

/string/

will find the next occurrence of *string*. It often happens that this is not the desired line, so the search must be repeated. This can be done simply by typing

//

This shorthand stands for “the most recently used context search expression.” It can also be used as the first string of the substitute command, as in

/string1/s//string2/

which will find the next occurrence of *string1* and replace it by *string2*. This can save a lot of typing. Similarly,

??

means “scan backwards for the same expression.”

1.14 CHANGE [c] AND INSERT [i]

“Change”, written as

c

is used to replace a number of lines with different lines, which are typed in at the terminal. For example, to change lines *.+1* through *\$* to something else, type

+.1,\$c
... type the lines of text you want here ...
.

The lines you type between the **c** command and the *.* will take the place of the original lines between start line and end line. This is most useful in replacing a line or several lines that have errors in them.

If only one line is specified in the **c** command, then just that line is replaced. (You can type in as many replacement lines as you like.) Notice the use of *.* (dot) to end the input; this works just like the *.* in the append command and must appear by itself on a new line. If no line number is given, line dot is replaced. The value of dot is set to the last line you typed.

“Insert” is similar to append; for instance,


```

/string/i
... type the lines to be inserted here ...
.

```

will insert the given text before the next line that contains *string*. The text between *i* and *.* is inserted before the specified line. If no line number is specified, dot is used. Dot is set to the last line inserted.

1.14.1 Exercise 7

“Change” is rather like a combination of delete followed by insert. Experiment to verify that

```

start, end d
i
... text..
.

```

is almost the same as

```

start, end c
... text..
.

```

These are not precisely the same if line \$ gets deleted. Check this out. What is dot?

Experiment with **a** and **i**, to see that they are similar, but not the same. You will observe that

```

line-number a
... text...
.

```

appends after the given line, while

```

line-number i
... text...
.

```

inserts before it. If no line number is given, **i** inserts before line dot, while **a** appends after line dot.

1.15 MOVING TEXT [m]

The move command **m** is used for cutting and pasting; it lets you move a group of lines from one place to another in the buffer. Suppose you want to move the first three lines of the buffer to the end of the buffer. You could do it by saying:

```

1,3w temp
$r temp
1,3d

```

but you can do it a lot easier with the **m** command:

1,3m\$

The general case is

start line, end line m after this line

Notice that there is a third line specified — where to move the text. Of course, the lines to be moved can be specified by context searches; if you had

First paragraph

...

end of first paragraph.

Second paragraph

...

end of second paragraph.

you could reverse the two paragraphs:

/Second/,/end of second/m/First/-1

Notice the **-1**: the moved text goes after the line mentioned. Dot gets set to the last line moved.

1.16 THE GLOBAL COMMANDS [g, v]

The global command **g** is used to execute one or more **ed** commands on all the lines in the buffer that match some specified string. For example,

g/peling/p

prints all lines that contain “peling”. More usefully,

g/peling/s//pelling/gp

makes the substitution everywhere on the line, then prints each corrected line. Compare this to

1,\$s/peling/pelling/gp

which only prints the last line substituted. Another subtle difference is that the **g** command does not return a **?** if, for example, “peling” is not found. In the same circumstances, the **s** command will return a question mark.

There may be several commands (including **a**, **c**, **i**, **r**, **w**, but not **g**); in that case, every line except the last must end with a backslash ****:

g/xxx/.-1s/abc/def/N

.+2s/ghi/jkl/N

.-2,.p

makes changes in the lines before and after each line that contains “xxx”, then prints all three lines.

The **v** command is the same as **g**, except that the commands are executed on every line that does *not* match the string following **v**:

`v/ /d`

deletes every line that does not contain a blank.

1.17 SPECIAL CHARACTERS

You may have noticed unexpected results when you used such characters as `.`, `*`, `$`, and others in context searches and the substitute command. The reason is rather complex, although the cure is simple. Basically, `ed` treats these characters as special, with special meanings. For instance, in a context search or the first string of the substitute command only, `.` means "any character," not a period, so

`/x.y/`

means "a line with an x, any character, and a y," not just "a line with an x, a period, and a y." The following special characters can cause problems if not used correctly.

`^ . $ [* \`

Note: The backslash character `\` is special to `ed`. If possible, avoid using it.

If you have to use one of the special characters in a substitute command, you can turn off its special meaning temporarily by preceding it with the backslash. Thus

`s/\\\.*/backslash dot star/`

will change `\.*` into "backslash dot star".

Here is a synopsis of the other special characters. First, the circumflex `^` signifies the beginning of a line. Thus

`/^string/`

finds *string* only if it is at the beginning of a line; it will find "string of pearls" but not "the string handler". The dollar sign `$` is just the opposite of the circumflex; it means the end of a line:

`/string$/`

will only find an occurrence of *string* that is at the end of some line. This implies, of course, that

`/^string$/`

will find only a line that contains just *string*, and

`/^.$/`

finds a line containing exactly one character.

The character `.`, as we mentioned above, matches anything;

```
/x.y/
```

matches any of

```
x+y
x-y
x y
x.y
```

This is useful in conjunction with `*`, which is a repetition character; `a*` is a shorthand for “any number of `a`’s,” so `.*` matches any number of any characters. This is used like this:

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

which changes an entire line, or

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

which deletes all characters in the line up to and including the last comma. (Since `.*` finds the longest possible match, this goes up to the last comma.)

Brackets are used to delimit “character classes”; for example,

```
/[0123456789]/
```

matches any single digit; any one of the characters inside the braces will cause a match. This can be abbreviated to `[0-9]`.

Finally, the `&` is another shorthand character. It is used only on the right-hand part of a substitute command where it means “whatever was matched on the left-hand side”. It is used to save typing. Suppose the current line contained

```
Now is the time
```

and you wanted to put parentheses around it. You could just retype the line, or you could say

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

using your knowledge of `^` and `$`. But the easiest way is to use the `&`:

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

This says “match the whole line, and replace it by itself surrounded by parentheses.” The `&` can be used several times in a line; consider using

```
s/./&? &!!/
```

to produce

```
Now is the time? Now is the time!!
```

You don’t have to match the whole line, of course. If the buffer contains

the end of the world
 you could type
/world/s//& is at hand/
 to produce

the end of the world is at hand

Examine this expression carefully. It illustrates how to take advantage of **ed** to save typing. The string */world/* found the desired line; the short-hand *//* found the same word in the line; and the **&** saves you from typing it again.

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

s/ampersand/\&/

will convert the word "ampersand" into its literal symbol **&** in the current line.

1.18 SUMMARY OF COMMANDS AND LINE NUMBERS

The general form of **ed** commands is the command name, perhaps preceded by one or two line numbers, and, in the case of **e**, **r**, and **w**, followed by a file name. Only one command is allowed per line, but a **p** command may follow any other command (except for **e**, **r**, **w** and **q**).

- a** Append, that is, add lines to the buffer (at line dot, unless a different line is specified). Appending continues until **.** is typed as the first character on a new line. Dot is set to the last line appended.
- c** Change the specified lines to the new text which follows. The new lines are terminated by a **.**, as with **a**. If no lines are specified, replace line dot. Dot is set to the last line changed.
- d** Delete the lines specified. If none are specified, delete line dot. Dot is set to the first undeleted line, unless **\$** is deleted, in which case dot is set to **\$**.
- e** Edit new file. Any previous contents of the buffer are thrown away, so issue a **w** beforehand.
- f** Print remembered filename. If a name follows **f**, the remembered name will be set to it.
- g** The command

g/---/commands

will execute the *commands* on those lines that contain **---**, which can be any context search expression.

- i** Insert lines before the specified line (or dot) until a **.** is typed as the first character on a new line. Dot is set to the last line inserted.
- m** Move lines specified to after the line named after **m**. Dot is set to the last line moved.
- p** Print specified lines. If none are specified, print line dot. A single line number is equivalent to line number **p**. A single return prints **.+1**, the next line.
- q** Quit **ed**. Wipes out all the text in the buffer if you enter it twice in a row without first entering a **w** command.
- r** Read a file into the buffer (at the end unless specified elsewhere.) Dot is set to the last line read.
- s** The command
 s/string1/string2/
 substitutes the characters *string1* into *string2* in the specified lines. If no lines are specified, the command makes the substitution in line dot. Dot is set to be the last line in which a substitution took place, which means that if no substitution took place, dot is not changed. **s** changes only the first occurrence of **string1** on a line; to change all of them, type a **g** after the final slash.
- v** The command
 v/---/commands
 executes *commands* on the lines that do not contain **---**.
- w** Writes out the buffer onto a file. Dot is not changed.
- .=** Print value of dot. **==(** by itself prints the value of **\$.**)
- !** The line
 !command-line
 causes the *command-line* to be passed to the Shell and executed.
- /.../** Context search, which searches for the next line that contains this string of characters, and prints it. Dot is set to the line where the string was found. Search starts at **.+1**, wraps around from **\$** to **1**, and continues to dot, if necessary.
- ?...?** Context search in reverse direction; starts search at **.-1**, scans to **1**, and wraps around to **\$**.

Chapter 2: The `ex` Reference Manual

2.1 INTRODUCTION

`Ex` is a line-oriented text editor that supports both command- and display-oriented editing. This chapter describes the command-oriented part of `ex`; the display editing features of `ex` are described in Chapter 3 of this section, *An Introduction to Display Editing with Vi*.

This chapter is based on the original `ex` reference manual written at the University of California at Berkeley.

2.2 USAGE

Each version of the editor has a set of options, which you can customize. The command `edit` invokes a version of `ex` designed for more casual or beginning users by changing the default settings of some of these options. To simplify the description which follows we assume the default settings of the options.

When invoked, `ex` determines the terminal type from the `TERM` variable in the environment. If there is a `TERMCAP` variable in the environment, and the type of the terminal described there matches the `TERM` variable, then that description is used. Also, if the `TERMCAP` variable contains a pathname (beginning with a `/`), then the editor will seek the description of the terminal in that file (rather than the default `/etc/termcap`.) If there is a variable `EXINIT` in the environment, then the editor will execute the commands in that variable; otherwise, if there is a file `.exrc` in your `HOME` directory, `ex` reads commands from that file, simulating a `source` command. Option setting commands placed in `EXINIT` or `.exrc` will be executed before each editor session.

A command to enter `ex` has the following prototype.

```
ex [-] [-v] [-t tag] [-r] [-l] [-wn] [-R] [+command] filename(s)
```

The most common case edits a single file with no options.

```
ex filename
```

The `-` command line option suppresses all interactive-user feedback. It is useful when processing editor scripts in command files. The `-v` option is equivalent to using `vi` rather than `ex`. The `-t` option is equivalent to an initial `tag` command, editing the file containing the `tag` and positioning the editor at its definition. The `-r` option is used in recovering after an editor or system crash, retrieving the last saved version of the named file or, if no file is specified, typing a list of saved files. The `-l` option sets up for editing LISP. It sets the `showmatch` and `lisp` options. The `-w`

option sets the default window size to n , and is useful on dialups to start in small windows. The **-R** option sets the **readonly** option at the start. *filename* arguments indicate files to be edited. An argument of the form **+command** indicates that the editor should begin by executing the specified command. If *command* is omitted, then it defaults to "\$", positioning the editor at the last line of the first file initially. Other useful commands here are scanning patterns of the form **/pat** or line numbers, e.g. **+100** starting at line 100.

2.3 FILE MANIPULATION

2.3.1 Current File

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

The current file is almost always considered to be *edited*. This means that the contents of the buffer are logically connected with the current file name, so that writing the current buffer contents onto that file, even if it exists, is a reasonable action. If the current file is not edited, then **ex** will not write on it, if it already exists.

Note: The *file* command will say "[Not edited]" if the current file is not considered edited.

2.3.2 Alternate File

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

2.3.3 Filename Expansion

Filenames within the editor may be specified using the normal Shell expansion conventions. In addition, the character **%** in filenames is replaced by the *current* file name and the character **#** by the *alternate* file name.

Note: This feature 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.

2.3.4 Multiple Files and Named Buffers

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

For saving blocks of text while editing, and especially when editing more than one file, *ex* has a group of named buffers. These are similar to the normal buffer, except that only a limited number of operations are available on them. The buffers have names *a* through *z*.

Note: It is also possible to refer to *A* through *Z*; the upper case buffers are the same as the lower, but commands append to named buffers rather than replacing when upper case names are used.

2.3.5 Read Only

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

2.4 EXCEPTIONAL CONDITIONS

2.4.1 Errors and Interrupts

When an error occurs, *ex* (optionally) rings the terminal bell and prints an error diagnostic. If the primary input is from a file, editor processing will terminate. If an interrupt signal is received, *ex* prints "Interrupt" and returns to its command level. When the primary input is a file, then *ex* will exit.

2.4.2 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) will attempt to preserve the buffer. The next time you log in you should be able to recover the work you were doing. At most, you will lose 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. If you were editing the file

resume, then you should change to the directory where you were when the crash occurred, giving the command

```
ex -r resume
```

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

You should get mail from the system telling you when a file has been saved after a crash. The command

```
ex -r
```

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

2.5 EDITING MODES

Ex has five distinct modes. Of these, “command” mode is most often used. Commands are entered in command mode when a ‘:’ prompt is present, and are executed each time a complete line is sent. In “text input” mode, **ex** gathers input lines and places them in the file. The **append**, **insert**, and **change** commands use text input mode. No prompt is printed when you are in text input mode. This mode is left by typing a ‘.’ alone at the beginning of a line, and *command* mode resumes.

The last three modes are **open** and **visual** modes, entered by the commands of the same name, and, within open and visual modes “text insertion” mode. Open and visual modes are described in Chapter 3 of this section.

2.6 COMMAND STRUCTURE

Most **ex** command names are English words, and the initial prefixes of the words are acceptable abbreviations. Any ambiguity in abbreviations is resolved in favor of the more commonly used commands. As an example, the command **substitute** can be abbreviated **s** while the shortest available abbreviation for the **set** command is **se**.

2.7 COMMAND PARAMETERS

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

Some commands take other information or parameters, which are always given after the command name. Examples of this would be option names in a *set* command i.e. “set number”, a file name in an *edit* command, a

regular expression in a *substitute* command, or a target address for a *copy* command, i.e. "1,5 copy 25".

2.7.1 Command Variants

A number of commands have two distinct variants. The variant form of the command is invoked by placing an "!" immediately after the command name. Some of the default variants may be controlled by options; in this case, the "!" serves to toggle the default.

2.7.2 Flags After Commands

The characters "#", "p", and "l" may be placed after many commands.

Note: A "p" or "l" must be preceded by a blank or tab except in the single special case "dp".

The operation specified by any of these characters will be executed after the command completes. Since **ex** normally prints the new current line after each change, the trailing "p" is rarely necessary. Any number of "+" or "-" characters may 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.

2.7.3 Comments

The "begin comment" character is the double quote: ". Any **ex** command line beginning with " is ignored. Comments beginning with " may also be placed at the ends of commands, except in cases where they could be confused as part of text (Shell escapes and the substitute and map commands).

2.7.4 Multiple Commands per Line

To place multiple commands on a line, separate each pair of commands with the "|" character.

Note: The "global" commands, comments, and the Shell escape "!" must be the last command on a line, as they are not terminated by a '|.

2.7.5 Reporting Large Changes

Most commands which change the contents of the editor buffer give feedback if the scope of the change exceeds a threshold given by the **report** option. This feedback helps to detect undesirably large changes, so that they may be quickly and easily reversed with an **undo**. After commands 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 **report** threshold.

2.8 COMMAND ADDRESSING

In this section, we summarize **ex** command addressing.

2.8.1 Addressing Primitives

- . (dot) The current line. Most commands leave the current line as the last line which they affect. The default address for most commands is the current line, thus “dot” is rarely used alone as an address.
- n* The *n*th line in the editor’s buffer, lines being numbered sequentially from 1.
- \$ The last line in the buffer.
- % An abbreviation for 1,\$ (i.e., the entire buffer).
- +*n* -*n* An offset relative to the current buffer line. The forms **.+3 +3** and **+++** are all equivalent. If the current line is line 100, all address line 103.
- /*pat*/ Scan forward for a line containing *pat*. *Pat* may be a string or a regular expression (as defined below).
- ?*pat*? Scan backward for a line containing *pat*. *Pat* may be a string or a regular expression (as defined below).

Note: The scans normally wrap around the end of the buffer. If all that is desired is to print the next line containing *pat*, then the trailing / or ? may be omitted. If *pat* is omitted or explicitly empty, then the last regular expression specified is located. The forms **\/** and **\?** scan using the last regular expression used in a scan; after a substitute, **//** and **??** would scan using the substitute’s regular expression.

- ‘ ‘ ‘*x* Before each non-relative motion of the current line ‘.’, the previous current line is marked with a tag, subsequently referred to as ‘ ‘ ‘. This makes it easy to refer or return to this previous context. Marks may also be established by the **mark** command, using single lower case letters *x* and the marked lines referred to as ‘ ‘ ‘*x*’.

2.8.2 Combining Addressing Primitives

Addresses to commands consist of a series of addressing primitives, separated by ‘,’ or ‘;’. Such address lists are evaluated left-to-right. When addresses are separated by ‘;’ the current line ‘.’ is set to the value of the previous addressing expression before the next address is interpreted. If more addresses are given than the command requires, then all but the last one or two are ignored. If the command takes two addresses, the first addressed line must precede the second in the buffer. Null address specifications are permitted in a list of addresses, the default

in this case is the current line '.'; thus ',100' is equivalent to ',,100'.

Note: It is an error to give a prefix address to a command which expects none.

2.9 COMMAND DESCRIPTIONS

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

address command ! parameters count flags

All parts are optional. When use without arguments, the command prints the next line in the file. When used within "visual" (**vi**) mode, **ex** ignores a ":" preceding any command.

In the following command descriptions, the default addresses are shown in parentheses. These parentheses are not part of the command. Abbreviations, where allowed, are shown at the beginning of the description, in brackets, as in [**ab**]. The brackets are not part of the abbreviation.

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

(**.**) **append** [**a**] Reads the input text and places it after the specified line. After the command, '.' addresses the last line input or the specified line if no lines were input. If address '0' is given, text is placed at the beginning of the buffer.

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

args The members of the argument list are printed, with the current argument delimited by '[' and ']'.
 [**]** Reads the input text and places it after the specified line. After the command, '.' addresses the last line input or the specified line if no lines were input. If address '0' is given, text is placed at the beginning of the buffer.

(**.,.**) **change count** [**c**] Replaces the specified lines with the input *text*. The current line becomes the last line input; if no lines were input it is left as for a **delete**.

c! The variant toggles **autoindent** during the **change**.

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

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

[d] Removes the specified lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end, the new last line becomes the current line. If a named *buffer* is specified by giving a letter, then the specified lines are saved in that buffer or appended to it if an upper case letter is used.

edit *file*

[e], [ex] Used to begin an editing session on a new *file*. The editor first checks to see if the buffer has been modified since the last **write** command was issued. If it has been, a warning is issued and the command is aborted. The command otherwise deletes the entire contents of the editor buffer, makes the named file the current file and prints the new filename. After ensuring that this file is an ASCII file, the editor reads the file into its buffer.

If the file is read without error, the number of lines and characters read is typed. If there were any non-ASCII characters in the file, they are stripped of their non-ASCII high bits, and any null characters in the file are discarded. If none of these errors occurred, the file is considered *edited*. If the last line of the input file is missing the trailing newline character, it will be supplied and a complaint will be issued. This command leaves the current line '.' at the last line read. If executed from within *open* or *visual*, the current line is initially the first line of the file.

e! *file*

The variant form of **edit** suppresses the complaint about modifications having been made and not written from the editor buffer, thus discarding all changes which have been made before editing the new file.

e+n *file*

Causes the editor to begin at line *n* rather than at the last line; *n* may also be an editor command containing no spaces, e.g.: **+ /pat/**.

file

[f] Prints the following: the current file name; whether it has been '[Modified]' since the last **write** command; whether it is *read only*; the

current line; the number of lines in the buffer; and the percentage of the way through the buffer of the current line. In the rare case that the current file is '[Not edited]', this is noted also; in this case, you have to use the form **w!** to write to the file, since the editor is not sure that a **write** won't destroy some *file* unrelated to the current contents of the buffer.

file *file*

The current file name is changed to *file* which is considered '[Not edited]'.

(1 , \$) **global** *pat cmds*

[g] First marks each line among those specified which matches the given regular expression. Then the given command list is executed with "." initially set to each marked line.

The command list consists of the remaining commands on the current input line and may continue to multiple lines by ending all but the last such line with a "\". If *cmds* (and possibly the trailing / delimiter) is omitted, each line matching *pat* is printed. **Append**, **insert**, and **change** commands and associated input are permitted; the "." terminating input may be omitted if it would be on the last line of the command list. *Open* and *visual* commands are permitted in the command list and take input from the terminal.

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

g! /*pat cmds*

[v] The variant form of **global** runs *cmds* at each line not matching *pat*.

- (.) **insert** [i] Places the given text before the specified line. The current line is left at the last line input; if there is no input, it is left at the line before the addressed line. This command differs from **append** only in the placement of text.
- i!** The variant toggles **autoindent** during the **insert**.
- (. , .+1) **join count flags** [j] Places the text from a specified range of lines together on one line. White space is adjusted at each junction to provide at least one blank character, two if there was a "." at the end of the line, or none if the first following character is a ")". If there is already white space at the end of the line, then the white space at the start of the next line will be discarded.
- j!** The variant causes a simple **join** with no white space processing; the characters in the lines are simply concatenated.
- (.) **k x** The **k** command is a synonym for **mark**. It does not require a blank or tab before the following letter.
- (. , .) **list count flags** Prints the specified lines in a more unambiguous way: tabs are printed as "↑I" and the end of each line is marked with a trailing "\$". The current line is left at the last line printed.
- map lhs rhs** The **map** command is used to define macros for use in *visual* mode. *Lhs* should be a single character, or the sequence "#n", where *n* is a digit, referring to function key *n*. When this character or function key is typed in *visual* mode, it will be as though the corresponding *rhs* had been typed. On terminals without function keys, you can type "#n".
- (.) **mark x** Gives the specified line mark *x*, a single lower case letter. The *x* must be preceded by a blank or a tab. The addressing form "'x" then addresses this line. The current line is not affected by this command.
- (. , .) **move addr** [m] The **move** command repositions the specified lines to be after *addr*. The first of the moved lines becomes the current line.

- next** [n] The next file from the command line argument list is edited.
- n!** The variant suppresses warnings about the modifications to the buffer not having been written out, discarding (irretrievably) any changes which may have been made.
- n *filelist***
- n +*command filelist*** The specified *filelist* is expanded and the resulting list replaces the current argument list; the first file in the new list is then edited. If *command* is given (it must contain no spaces), then it is executed after editing the first such file.
- (. . .) *number count flags* [# or nu]** Prints each specified line preceded by its buffer line number. The current line is left at the last line printed.
- (.) *open flags***
- (.) *open/pat/flags*** [o] Enters intraline editing **open** mode at each addressed line. If *pat* is given, then the cursor will be placed initially at the beginning of the string matched by the pattern. To exit this mode use **Q**. See Chapter 3 of this section for more details.
- preserve** The current editor buffer is saved as though the system had just crashed. This command is for use only in emergencies when a **write** command has resulted in an error and you don't know how to save your work. After a **preserve**, you should seek help from a system administrator.
- (. . .) *print count*** [p or P] Prints the specified lines with non-printing characters printed as control characters " $\uparrow x$ "; delete (octal 177) is represented as " $\uparrow ?$ ". The current line is left at the last line printed.
- (.) *put buffer*** [pu] Puts back previously **deleted** or **yanked** lines. Normally, used with **delete** to effect movement of lines, or with **yank** to effect duplication of lines. If no *buffer* is specified, then the last deleted or yanked text is restored, but no modifying commands may intervene between the **delete** or **yank** and the **put**, nor may lines be moved between files without using a named buffer. By using

a named buffer, text may be restored that was saved there at any previous time.

quit

[q] Causes **ex** to terminate. No automatic write of the editor buffer to a file is performed. However, **ex** issues a warning message if the file has changed since the last **write** command was issued, and does not **quit**. **Ex** will also issue a diagnostic if there are more files in the argument list. Normally, you want to save your changes, and you should give a **write** command; if you want to discard them, use the **q!** command variant.

q!

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

(.) **read file**

[r] Places a copy of the text of the given file in the editing buffer after the specified line. If no *file* is given, the current file name is used. The current file name is not changed unless there is none, in which case, *file* becomes the current name. The sensibility restrictions for the **edit** command apply here also. If the file buffer is empty and there is no current name, then **ex** treats this as an **edit** command. Address "0" is legal for this command and causes the file to be read at the beginning of the buffer. Statistics are given as for the **edit** command when the **read** successfully terminates. After a **read**, the current line is the last line read. Within **open** and **visual**, the current line is set to the first line read rather than the last.

(.) **read !command**

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

recover file

Recovers *file* from the system save area. Used after a system crash or accidental hangup. Note that the system saves a copy of the file you were editing only if you have made changes to the file. If you are using *preserve*, you will be notified by mail when a file is saved.

- rewind** [rew] The argument list is rewound, and the first file in the list is edited.
- rew!** Rewinds the argument list discarding any changes made to the current buffer.
- set *parameter*** With no arguments, prints those options whose values have been changed from their defaults; with parameter **all**, it prints all of the option values.
- Giving an option name followed by a “?” causes the current value of that option to be printed. The “?” is unnecessary unless the option is Boolean valued. Boolean options are given values either by the form “set *option*” to turn them on or “set *nooption*” to turn them off; string and numeric options are assigned via the form “set *option*=value”.
- More than one parameter may be given to *set* ; parameters are interpreted left-to-right.
- shell** [sh] A new Shell is created. When it terminates, editing resumes.
- source *file*** [so] Reads and executes commands from the specified file. **Source** commands may be nested.
- (. . .) substitute /*pat*/*repl*/ *options count flags***
- [s] On each specified line, the first instance of pattern *pat* is replaced by replacement pattern *repl*. If the **global** indicator option character **g** appears, then all instances are substituted; if the **confirm** indication character **c** appears, then before each substitution the line to be substituted is typed with the string to be substituted marked with ‘^’ characters. By typing a **y**, you can cause the substitution to be performed. Any other input results in no change. After a **substitute** the current line is the last line substituted.
- Lines may be split by substituting newline characters into them. The newline in *repl* must be escaped by preceding it with a \. Other metacharacters available in *pat* and *repl* are described below.

stop

Suspends the editor, returning control to the top level Shell. If **autowrite** is set and there are unsaved changes, a write is done first unless the form **stop!** is used.

(. . .) substitute options count flags

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

(. . .) t addr flags

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

ta tag

The focus of editing switches to the location of *tag*, switching to a different line in the current file where it is defined, or if necessary to another file.

Note: If you have modified the current file before giving a *tag* command, you must write it out; giving another *tag* command, specifying no *tag* will reuse the previous tag.

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

unabbreviate word

[**una**] Delete *word* from the list of abbreviations.

undo

[**u**] Reverses the changes made in the buffer by the last buffer editing command. Note that **global** commands are considered a single command by **undo** (as are **open** and **visual**.) Also, the commands **write** and **edit** which interact with the file system cannot be undone. **Undo** is its own inverse.

Undo always marks the previous value of the current line "." as "' '". After an **undo** the

current line is the first line restored or the line before the first line deleted if no lines were restored. For commands with more global effect such as **global** and **visual**, the current line regains its pre-command value after an **undo**.

unmap *lhs*

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

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

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

version

[**ve**] Prints the current version number of the editor as well as the date the editor was last changed.

(.) **visual** *type count flags*[**vi**] Enters visual mode at the specified line. *Type* is optional and may be “_”, “^” or “.” as in the **z** command to specify the placement of the specified line on the screen. By default, if *type* is omitted, the specified line is placed as the first on the screen. A *count* specifies an initial window size; the default is the value of the option *window*. See Chapter 3 of this section, *An Introduction to Display Editing with Vi*, for more details. To exit this mode, type **Q**.

visual *file*

Also **visual** +*n file* From visual mode, this command is the same as **edit**.

(1 , \$) **w** *file*

[**w**] Writes changes made back to *file*, printing the number of lines and characters written. When *file* is omitted, the text is written back to the file named when the editor was originally invoked. If a *file* is specified, then text will be written to that file.

Note: The editor writes to a file only if it is the current file and is *edited*, if the file does not exist, or if the file is actually a teletype, */dev/tty*, */dev/null*. Otherwise, you must give the variant form **w!** to force the write.

If the file does not exist it is created. The current file name is changed only if there is no current file name; the current line is never

changed.

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

(1 , \$) **w**rite>> *file*

[**w**>>] Writes the buffer contents at the end of an existing *file*.

w! *name*

Overrides the checking of the normal **w**rite command, and will write to any file which the system permits.

(1 , \$) **w !***command*

Writes the specified lines into **command**.

Note: The space between **w** and **!** is important. It's the only difference between **w!**, which overrides checks and **w !**, which writes to a command.

wq *name*

Similar to a **w**rite and then a **quit** command.

wq! *name*

The variant overrides checking on the sensibility of the **w**rite command, as **w!** does.

xit *name*

If any changes have been made and not written, **xit** writes the buffer out, then quits.

(. , .) **y**ank *buffer count*

[**ya**] Places the specified lines in the named *buffer*, for later retrieval via **put**. If no buffer name is specified, the lines go to a more volatile place; see the **put** command description.

(.+1) **z** *count*

Print the next *count* lines, default **count** = **window**.

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

Prints a window of text with the specified line at the top. If *type* is ‘-’ the line is placed at the bottom; a ‘.’ causes the line to be placed in the center. A count gives the number of lines to be displayed rather than double the number specified by the *scroll* option. On a CRT, the screen is cleared before display begins unless **count** < *screen size*. The current line is left at the last line printed.

Note: Forms “z=” and “z^” also exist; “z=” places the current line in the center, surrounds it with lines

of “-” characters and leaves the current line at this line. The form “z^” prints the window before “z-” would. The characters “+”, “^”, and “-” may be repeated for cumulative effect.

! *command*

The remainder of the line after the ! character is sent to a Shell to be executed. Within the text of *command*, the characters “%” and “#” are expanded as in filenames and the character “!” is replaced with the text of the previous command. Thus, in particular, “!!” repeats the last such Shell escape. If any such expansion is performed, the expanded line will be echoed. The current line is unchanged by this command. If there has been “[No write]” of the buffer contents since the last change to the editing buffer, then a diagnostic will be printed before the command is executed as a warning. A single “!” is printed when the command completes.

(*addr* , *addr*) ! *command* Takes the specified address range and supplies it as standard input to *command*; the output of *command* replaces the input lines.

(\$) = Prints the line number of the addressed line. The current line is unchanged.

(. , .) > *count flags*

(. , .) < *count flags*

Does an intelligent shift of 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-blank characters are discarded in a left shift. The current line becomes the last line which changed due to the shifting.

↑Z

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.

(.+1 , .+1)

An address alone causes the addressed lines to be printed. A blank line prints the next line in the file.

`(.,.) & options count flags`

Repeats the previous *substitute* command.

`(.,.) ~ options count flags`

Replaces the previous regular expression with the previous replacement pattern from a substitution.

2.10 REGULAR EXPRESSIONS

2.10.1 Regular Expressions

A regular expression specifies a set of strings of characters. A member of this set of strings is said to be *matched* by the regular expression. **Ex** remembers two previous regular expressions: the previous regular expression used in a *substitute* command and the previous regular expression used elsewhere (referred to as the previous *scanning* regular expression.) The previous regular expression can always be referred to by a null *re*, e.g., “//” or “??”.

2.10.2 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 the discussion of regular expressions assumes that the setting of this option is *magic*.

Note: To discern what is true with *nomagic* it suffices to remember that the only special characters in this case will be “^” at the beginning of a regular expression, “\$” at the end of a regular expression, and “\”. With *nomagic* the characters “~” and “&” also lose their special meanings related to the replacement pattern of a substitute.

2.10.3 Regular Expression Summary

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

char An ordinary character matches itself. The characters “^” at the beginning of a line, “\$” at the end of line, “*” as

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

2.10.4 Combining Regular Expression Primitives

The concatenation of two regular expressions matches the leftmost and then longest string which can be divided with the first piece matching the first regular expression and the second piece matching the second. Any of the (single character matching) regular expressions mentioned above may be followed by the character "*" to form a regular expression which matches any number of adjacent occurrences (including 0) of characters matched by the regular expression it follows.

The tilde (~) character may be used in a regular expression, and matches the text which defined the replacement part of the last *substitute* command. A regular expression may be enclosed between the sequences "\(" and "\)" with side effects in the *substitute* replacement patterns.

2.10.5 Substitute Replacement Patterns

The basic metacharacters for the replacement pattern are "&" and "~"; these are given as "\&" and "\~" when *nomagic* is set. Each instance of "&" is replaced by the characters which the regular expression

matched. The metacharacter “~” stands, in the replacement pattern, for the defining text of the previous replacement pattern.

Other metasequences possible in the replacement pattern are always introduced by the escaping character “\”. The sequence “\n” is replaced by the text matched by the *n*th regular subexpression enclosed between “(” and “)”.

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

The sequences “\u” and “\l” cause the following character in the replacement to be converted to upper- or lower-case respectively if this character is a letter. The sequences “\U” and “\L” turn such conversion on, either until “\E” or “\e” is encountered, or until the end of the replacement pattern.

2.11 OPTION DESCRIPTIONS

In the following descriptions, the full name of the option (bold type) is followed by the option’s default value. The abbreviation appears in brackets. The brackets are not part of the abbreviation.

autoindent noai

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

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

A line with no characters added to it turns

into a completely blank line (the white space provided for the **autoindent** is discarded.) Lines beginning with an “↑” and immediately followed by a ↑**D** cause the input to be repositioned at the beginning of the line, but retain the previous indent for the next line. Similarly, a “0” followed by a ↑**D** repositions at the beginning but without retaining the previous indent.

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

autoprint ap

[ap] Causes the current line to be printed after each **delete**, **copy**, **join**, **move**, **substitute**, **t**, **undo** or **shift** command. This has the same effect as supplying a trailing **p** to each such command. **Autoprint** is suppressed in **globals**, and only applies to the last of many commands on a line.

autowrite noaw

[aw] Causes the contents of the buffer to be written to the current file if you have modified it and gives a **next**, **rewind**, **stop**, **tag**, or **!** command, or a “^↑” (switch files) or “^]” (tag goto) command in **visual**.

Note: The **edit** and **ex** commands do **not** autowrite. In each case, there is an equivalent way of switching when autowrite is set to avoid the **autowrite**.

beautify nobeautify

[bf] Causes all control characters except tab, newline, and form-feed to be discarded from the input. A complaint is registered the first time a backspace character is discarded. **Beautify** does not apply to command input.

directory dir=/tmp

[dir] Specifies the directory in which **ex** places its buffer file. If this directory is not writable, then the editor will exit abruptly when it is unable to create its buffer there.

edcompatible noedcompatible

Causes the presence or absence of **g** and **c** suffixes on substitute commands to be remembered, and to be toggled by repeating the suffixes. The suffix **r** makes the substitution

- be as in the `~` command, instead of like `&`.
- errorbells** noed
[eb] Error messages are preceded by a bell.
- Note:** Bell ringing in *open* and *visual* on errors is not suppressed by setting **noeb**.
- If possible, the editor places the error message in a standout mode of the terminal (such as inverse video) instead of ringing the bell.
- hardtabs** ht=8
[ht] Gives the boundaries on which terminal hardware tabs are set (or on which the system expands tabs).
- ignorecase** noic
[ic] All upper-case characters in the text are mapped to lower-case in regular expression matching. In addition, all upper case characters in regular expressions are mapped to lower case except in character class specifications.
- lisp** nolisp
Autoindent indents appropriately for LISP code, and the `() { } [[and]]` commands in **open** and **visual** are modified to have meaning for LISP.
- list** nolist
All printed lines are displayed showing tabs and end-of-lines as in the **list** command.
- magic** magic
If **nomagic** is set, the number of regular expression metacharacters is greatly reduced, with only `^` and `$` having special effects. In addition, the metacharacters `~` and `&` of the replacement pattern are treated as normal characters. All the normal metacharacters may be made **magic** when **nomagic** is set by escaping them with a backslash (`\`).
- mesg** mesg
If **nomesg** is set, it inhibits other users from doing a **write** to your terminal while you are in visual mode.
- number** nonumber
[nu] Causes all output lines to be numbered. In addition, each input line will be prompted for its line number.
- open** open
If **noopen**, the commands **open** and **visual** are not permitted. This is set for **edit** to prevent confusion resulting from accidental entry to open or visual mode.

- optimize optimize** [**opt**] Throughput of text is expedited by setting the terminal to not do automatic carriage returns when printing more than one (logical) line of output, greatly speeding output on terminals without addressable cursors when text with leading white space is printed.
- paragraphs para=IPLPPPQPP LIbp**
- [**para**] Specifies the paragraphs for the { and } operations in **open** and **visual**. The pairs of characters in the option's value are the names of the macros used to start paragraphs.
- prompt prompt** Command mode input is prompted for with a "·".
- redraw noredraw** The editor simulates an intelligent terminal on a dumb terminal (e.g. during insertions in **visual** the characters to the right of the cursor position are refreshed as each input character is typed.) Useful only at very high speed.
- remap remap** If on, macros are repeatedly tried until they are unchanged. For example, if **o** is mapped to **O**, and **O** is mapped to **I**, then if *remap* is set, **o** will map to **I**, but if *noremap* is set, it will map to **O**.
- report report=5** Specifies a threshold for feedback from commands. Any command which modifies more than the specified number of lines will provide feedback as to the scope of its changes. For commands such as **global**, **open**, **undo**, and **visual** which have potentially more far reaching scope, the net change in the number of lines in the buffer is presented at the end of the command, subject to this same threshold. Thus, during a **global** operation, notification on the individual commands performed is suppressed.
- scroll scroll=1/2 window** Determines the number of logical lines scrolled when an end-of-file is received from a terminal input in command mode, and the number of lines printed by a command mode **z** command (double the value of **scroll**).
- sections sections=SHNHH HU**
- Specifies the section macros for the [[and]]

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

shell SHELL or sh=/bin/sh

[sh] Gives the path name of the Shell used by the Shell escape ! and by the **shell** command. The default is taken from the environment variable SHELL if present.

shiftwidth sw=8

[sw] Gives the width of a software tab stop, used in reverse tabbing with ↑D when using **autoindent** to append text, and by the shift commands.

showmatch nosm

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

slowopen

[slow] Affects the display algorithm used in **visual** mode, holding off display updating during input of new text to improve throughput when the terminal in use is both slow and unintelligent. See Chapter 3 of this section, *An Introduction to Display Editing with vi* for more details.

tabstop ts=8

[ts] The editor expands tabs in the input file to be on **tabstop** boundaries for the purposes of display.

taglength tl=0

[tl] Tags are not significant beyond this many characters. A value of zero (the default) means that all characters are significant.

tags

A path of files to be used as tag files for the **tag** command. A requested tag is searched for in the specified files, sequentially. By default, files *.tags* and */usr/lib/tags* are searched for.

term

The terminal type of the output device. Defaults to environment variable TERM if set.

terse noterse

Use shorter error diagnostics.

warn warn

Warn if there has been "[No write since last change]" before a "!" command escape.

- window** The number of lines in a text window in the **visual** command. The default is 8 at slow speeds (600 baud or less), 16 at medium speed (1200 baud), and the full screen (minus one line) at higher speeds.
- Note:** The "commands" **w300**, **w1200**, and **w9600** are not true options but set **window** only if the speed is slow (300), medium (1200), or high (9600), respectively. They are suitable for an EXINIT and make it easy to change the 8/16/full screen rule.
- wrapsan ws** [**ws**] Searches using the regular expressions in addressing will wrap around past the end of the file.
- wrapmargin wm=0** [**wm**] Defines a margin for automatic wrap-over of text during input in **open** and **visual** modes.
- writeany nowa** [**wa**] Inhibits the checks normally made before **write** commands, allowing a write to any file to which you have access.

2.12 LIMITATIONS

Editor limits you are likely to encounter are:

- 1024 characters per line,
- 256 characters per global command list,
- 128 characters per file name,
- 128 characters in the previous inserted and deleted text in **open** or **visual**,
- 100 characters in a Shell escape command,
- 63 characters in a string valued option,
- 30 characters in a tag name, and
- a limit of 250000 lines in the file is silently enforced.

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

C

C

C

C

C

Chapter 3: An Introduction to Display Editing With vi

3.1 INTRODUCTION

Vi (**v**isual) is a display-oriented interactive text editor. On DOMAIN systems, the shell window in which you invoke **vi** becomes a VT100 Emulator. The "screen" of this "terminal" acts as a window into the file that you are editing. Changes that you make to the file are reflected in what you see.

Vi lets you insert new text at any place in the file. Most of the commands to **vi** move the cursor around in the file. There are commands to move the cursor forward and backward in units of characters, words, lines, sentences and paragraphs. A small set of operators, like **d** for delete and **c** for change, combine with the motion commands to perform operations such as "delete word" or "change paragraph." This regularity and the mnemonic assignment of commands to keys makes the **vi** command set easy to remember and to use.

Vi works on a large number of display terminals, as well as on DOMAIN nodes. While it is advantageous to have an intelligent terminal which can locally insert and delete lines and characters from the display, **vi** also works well on dumb terminals that communicate over slow phone lines. The editor makes allowance for the low bandwidth in these situations and uses smaller window sizes and different display updating algorithms to make best use of the limited speed available.

It is also possible to use the **vi** command set and a one-line editing window on hardcopy terminals, storage tubes and "glass tty's." The full command set of the more traditional, line-oriented editor *ex* is available within **vi**; it is quite simple to switch between the two modes of editing.

This chapter is based on the original *Introduction to vi*, written at the University of California at Berkeley.

3.2 GETTING STARTED

This chapter provides a quick introduction to **vi**. (Pronounced *vee-eye*.) As you read this, run **vi** on a non-critical file with which you are familiar. In the first part of this chapter, we describe the fundamentals of using **vi**. Topics of less universal interest are presented in later sections.

This chapter includes a section that presents the special meanings that various keyboard characters have for **vi**.

3.2.1 Notational Conventions

In the examples we present, input that must be typed “as-is” will be set in **bold type**. Text which should be replaced with appropriate input will be given in *Italics*. We will represent special characters and keyboard keys in SMALL CAPITALS.

3.2.2 Vi and the VT100 Emulator Program

When run on a DOMAIN node, **vi** automatically invokes the */com/vt100* terminal emulation program. This program performs two major functions:

- It remaps the keyboard so that all VT100 function keys (including ESC and RUB) are supported.
- It “borrows” the shell window and, using graphics primitives, mimics the behavior of a VT132 terminal. (A VT132 is a VT100 with insert/delete character and insert/delete line capabilities.)
- It allows **vi** to communicate with this “terminal” using normal escape sequences. The *termcap* entry for an *apollo_19l* terminal is nearly the same as the one for a *vt132*.

While a real VT132 can only display 24 lines of 80 columns, the emulator will use as many lines and columns as will fit into the window in which it was invoked. If you need to use **vi**, we recommend that you first invoke a UNIX shell in a window of a convenient size, then dedicate that window to running **vi**.

3.2.3 Keyboard Mapping

The table below shows how the keys on a DOMAIN keyboard map to the keys of a VT100. This key mapping is only in effect when the cursor is in a window running **vi** or the **vt100** program.

Note: Key definitions marked with a † are for the 880 (high-profile) keyboard only.

DOMAIN keyboard key	VT100 keypad
<ESC> <INS MODE>† <CHAR DEL>	
<F2> <F3> <F4> <F5>	
<SHIFT/F2> <SHIFT/F3> <SHIFT/F4> <SHIFT/F5>	
<CTRL/F2> <CTRL/F3> <CTRL/F4> <CTRL/F5> <F6> <F7> SHIFT/<F6>	
SHIFT/<F7> CTRL/<F6> CTRL/<F7>	

3.2.4 Specifying Terminal Type

Note: If you only run **vi** on a DOMAIN node (if you never use a dumb terminal), you may skip this section.

If you are using a terminal connected to the DOMAIN system via hardwired or phone lines, you must tell the system what kind of terminal you are using before you invoke **vi**. Here is a partial list of terminal type codes. If your terminal does not appear here, see your System Administrator.

Code	Full name	Type
2621	Hewlett-Packard	2621A/P
2645	Hewlett-Packard	264x
act4	Microterm	ACT-IV
act5	Microterm	ACT-V
adm3a	Lear Siegler	ADM-3a
adm31	Lear Siegler	ADM-31
c100	Human Design Concept	100
dm1520	Datamedia	1520
dm2500	Datamedia	2500
dm3025	Datamedia	3025
fox	Perkin-Elmer	Fox
h1500	Hazeltine	1500
h19	Heathkit	h19
i100	Infoton	100
mime	Imitating a smart act4	Intelligent
t1061	Teleray	1061
vt52	Dec	VT52

Suppose, for example, that you have a Hewlett-Packard HP2621A terminal. The code used by the system for this terminal is "2621". To tell the system that you are using a 2621, use one of the following UNIX commands. In the C-Shell, say:

```
% setenv TERM 2621
```

If you are using a Bourne shell, type the commands:

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

If you want to arrange to have the terminal type set automatically when you log in, use the *tset* program. For example, if you dial in on a *VT52*, terminal, but also use a *DOMAIN* node at work, a typical line for your *.login* file (if you use the C-Shell) would be

```
setenv TERM `tset - -d vt52`
```

or for your *.profile* file (if you use the Bourne Shell)

```
TERM=`tset - -d vt52`
```

Tset knows when you are using a node, and needs only to be told that when you dial in, it will be talking to a *VT52*. Usually, *tset* is used to change the erase and kill characters, too.

3.2.5 Editing a File

After telling the system what kind of terminal you have, make a copy of a familiar file — one that is not too long — and run *vi* on this file, giving the command

```
% vi name
```

where *name* is the name of the copy file you just created. When you do this, the window will clear and the text of your file will appear in it.

Note: If you gave the system an incorrect terminal type code, then the editor may make a mess out of your screen. This happens when it sends control codes for one kind of terminal to some other kind of terminal. If this happens, hit the keys **:q** (colon and the q key) and then hit the RETURN key. This should get you back to the shell. Another possibility (if you don't see your file) is that you typed the wrong file name and **vi** has printed an error diagnostic. In this case, you should follow the above procedure for getting back to the Shell, then re-try the procedure. If the **vi** doesn't seem to respond to the commands which you type here, try sending an interrupt to it by typing **↑I**, and then hitting the **:q** command again followed by a carriage return.

3.2.6 The Buffer

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

3.2.7 View

If you want to use the editor to look at a file, rather than to make changes in it, invoke it as **view** instead of **vi**. This will set the *readonly* option which will prevent you from accidentally overwriting the file.

3.2.8 Arrow Keys

Vi supports the cursor positioning keys of most terminals. Whether or not you have cursor positioning keys, you can use the h, j, k, and l keys as cursor positioning keys.

Note: If you are using an HP2621 terminal, the function keys must be *shifted* to be read by **vi**, otherwise, they only act locally. Unshifted use will leave the cursor positioned incorrectly.

The h key command moves the cursor to the left (like **↑h** — the back-space), j moves down (in the same column), k moves up (in the same column), and l moves to the right.

3.2.9 Special Characters: ESC, RETURN and DEL

Look on your keyboard for a key labelled ESC or (if you're using a terminal) ALT. On DOMAIN low-profile keyboards, ESC is near the upper left corner of the character key area. On DOMAIN 880 keyboards, ESC is mapped to INS MODE when **vi** is running. Try hitting this key a few times. The editor will beep to indicate that it is in a quiescent state.

Note: On some terminals, **vi** will quietly flash the screen rather than ringing the bell.

Partially formed commands are cancelled by ESC, and when you insert text in the file, you end the text insertion with ESC. This key is a fairly harmless one to hit, so you can just hit it if you don't know what is going on until the editor rings the bell.

The RETURN key is important because it is used to terminate certain commands. On all DOMAIN keyboards (as well as on most terminals), RETURN is located at the right side of the keyboard, and is the same key used to terminate shell commands.

Another very useful key is the DEL or RUB key, which generates an interrupt, telling the editor to stop what it is doing.

Note: On DOMAIN nodes, the interrupt function is mapped to ↑I by the *unix_keys* key definition file.

It is a forceful way of making the editor listen to you, or to return it to the quiescent state if you don't know or don't like what is going on. Try hitting the “/” key on your terminal. This key is used when you want to specify a string to be searched for. The cursor should now be positioned after a “/” prompt at the bottom line of the window. You can get the cursor back to the current position by hitting the DEL or RUB key; try this now.

Note: Backspacing over the “/” will also cancel the search.

From now on we will simply refer to hitting the I (or DEL or RUB) key as “sending an interrupt.”

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

3.2.10 Getting Out of the Editor

After you have worked with this introduction for a while, and you wish to do something else, you can give the command **ZZ** to the editor. This will write the contents of the editor's buffer back into the file you are editing (if you made any changes), then quit **vi**. You can also end an editor session by giving the command **:q!RETURN**.

Note: All commands which read from the last display line can also be terminated with an ESC as well as a RETURN.

The **:q!** command ends the editor session and discards all the changes you've made since your last write (i.e., **:w**). You may need to use this command if, for example, you change the editor's copy of a file you wished only to view. **Don't** give this command when you really want to

save the changes you have made.

3.3 MOVING AROUND IN THE FILE

3.3.1 Scrolling and Paging

The most useful of the many scroll/page commands is generated by hitting the CTRL (Control) and D keys at the same time, a control-D or “↑D”. From now on, we will use this two-character notation when referring to control sequences. You may have a key labelled “^” on your terminal. This key will be represented as “^” in this book. The “↑” notation will be used only as part of the “↑X” notation for control characters.

As you know now if you tried hitting ↑D, this command scrolls down in the file. The D thus stands for down. For instance the command to scroll up is ↑U. Many dumb terminals can't scroll up at all, in which case hitting ↑U clears the screen and refreshes it with a line which is farther back in the file at the top.

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

There are other ways to move around in the file; the keys ↑F and ↑B move forward and backward a page, keeping a couple of lines of continuity between screens so that it is possible to read through a file using these rather than ↑D and ↑U if you wish.

Notice the difference between scrolling and paging. If you are trying to read the text in a file, hitting ↑F to move forward a page does not allow you to view lines that were on the previous page. Scrolling, on the other hand, leaves previous lines visible. You can continue to read the text as scrolling is taking place.

3.3.2 Searching, Goto, and Previous Context

Another way to position the cursor in the file is by giving the editor a string to search for. Type the character / followed by a string of characters terminated by RETURN. The editor will position the cursor at the next occurrence of this string. Try hitting n to then go to the next occurrence of this string. The character ? will make vi search backwards from where you are, and is otherwise like /.

Note: These searches will normally wrap around the end of the file, and thus find the string even if it is not on a line in the direction of your search, provided it is somewhere in the file. You can disable this wraparound in scans by giving the command `:se nowrapscanRETURN`, or its abbreviation `:se nowsRETURN`.

If the search string you give the editor is not present in the file, the editor will print a message in the last line of the window, and the cursor will be returned to its initial position.

If you wish the search to match only at the beginning of a line, begin the search string with a `^`. To match only at the end of a line, end the search string with a `$`. Thus `/^searchRETURN` will search for the word “search” at the beginning of a line, and `/last$RETURN` searches for the word “last” at the end of a line.

Note: **Vi** can search for a string that is a regular expression in the sense of the editors *ex(1)* and *ed(1)*. If you don't wish to learn about this yet, you can disable this more general facility by doing `:se nomagicRETURN`; by putting this command in `EXINIT` in your environment, you can have this always be in effect (more about `EXINIT` later.)

The command **G**, when preceded by a number will position the cursor at that line in the file. Thus `1G` will move the cursor to the first line of the file. If you give **G** no count, then it moves to the end of the file.

If, because you are near the end of the file, there are unused lines on the screen, the editor will place only the character “~ ” on those lines that are past the end of the file.

You can find out the state of the file you are editing by typing a `↑G`. The editor will show you the name of the file you are editing, the number of the current line, the number of lines in the buffer, and how far (in percentage of characters) you have moved through the buffer. Try doing this now, and remember the number of the line you are on. Give a **G** command to get to the end and then another **G** command to get back where you were.

You can also get back to a previous position by using the command ```` (two back quotes). This is often more convenient than **G** because it requires no advance preparation. Try giving a **G** or a search with `/` or `?` and then a ```` to get back to where you were. If you accidentally hit **n** or any command which moves you far away from a context of interest, you can quickly get back by hitting ````.

3.3.3 Moving Around on the Screen

Now try just moving the cursor around on the screen. If your terminal has arrow keys, try them and convince yourself that they work. If you don't have working arrow keys, you can always use **h**, **j**, **k**, and **l**. Experienced users of **vi** prefer using these keys since, unlike arrow keys, they don't require you to move your hand away from the character keys on the keyboard.

Hit the `+` key. Each time you do, notice that the cursor advances to the next line in the file, at the first nonblank position on the line. The `-` key is like `+` but goes the other way.

These are very common keys for moving up and down lines in the file. Notice that if you go off the bottom or top with these keys, then the screen will scroll as necessary to bring a line at a time into view. The RETURN key has the same effect as the + key.

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

3.3.4 Moving Within a Line

Now try picking a word on some line on the screen (not the first word on the line). Using RETURN and -, move the cursor to the line the word is in. Hit the **w** key. This advances the cursor to the next word on the line. Now hit the **b** key to back up, by words, in the line. Also try the **e** key which advances you to the end of the current word rather than to the beginning of the next word. Also try SPACE (the space bar) which moves right one character and the BS (backspace or ↑**H**) key which moves left one character. The key **h** works as ↑**H** does and is useful if you don't have a BS key. (Also, as noted just above, **l** will move to the right.)

If the line had any punctuation, you may have noticed that the **w** and **b** keys stopped at each group of punctuation. You can also go backwards and forwards words without stopping at punctuation by using **W** and **B** rather than the lower case equivalents. Think of these as bigger words. Try these on a few lines with punctuation to see how they differ from the lower case **w** and **b**.

These "word" movement keys wrap around the end of line, rather than stopping at the end. Try moving to a word on a line below where you are by repeatedly hitting **w**.

3.3.5 Summary of Cursor Movement and Scrolling

SPACE	advance the cursor one position
↑B	backwards to previous page
↑D	scrolls down in the file
↑E	exposes another line at the bottom
↑F	forward to next page
↑G	tell what is going on
↑H	backspace the cursor
↑N	next line, same column
↑P	previous line, same column
↑U	scrolls up in the file
↑Y	exposes another line at the top
+	next line, at the beginning
-	previous line, at the beginning
/	scan for a following string forwards
?	scan backwards
B	back a word, ignoring punctuation
G	go to specified line, last default
H	home screen line
M	middle screen line
L	last screen line
W	forward a word, ignoring punctuation
b	back a word
e	end of current word
n	scan for next instance of / or ? pattern
w	word after this word

3.4 MAKING SIMPLE CHANGES

3.4.1 Inserting

One of the most useful commands is the **i** (insert) command. After you type **i**, everything you type until you hit ESC is inserted into the file. Try this now; position yourself to some word in the file and try inserting text before this word. If you are using a dumb terminal, it will seem, for a minute, that some of the characters in your line have been overwritten, but they will reappear when you hit ESC.

Now find a singular noun that can be made plural by the addition of a final "s". Position the cursor at this word and type **e** (move to end of word), then **a** for append and then "sESC" to terminate the textual insert.

Try inserting and appending a few times to make sure you understand how this works; **i** placing text to the left of the cursor, **a** to the right.

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

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

If you need to type in more than one line of text, hit a RETURN at the end of any line you are typing. A new line will be created for text, and you can continue to type. On some terminals, vi editor may choose to wait before redrawing the lower portion of the screen. This will make it appear as though you are typing over existing screen lines, but it avoids the lengthy delay which would occur if the editor attempted to keep the tail of the screen always up to date. The tail of the screen will be fixed up, and the "overwritten" lines will reappear, when you hit ESC.

While you are inserting new text, you can use the characters you normally use at the system command level (usually ↑H or #) to backspace over the last character which you typed, and the character which you use to kill input lines (usually @, ↑X, or ↑U) to erase the input you have typed on the current line.

Note: The character ↑H (backspace) always works to erase the last input character here, regardless of what your erase character is.

The character ↑W will erase a whole word and leave the cursor after the space following the previous word; it is useful for quickly backing up in an insert.

Notice that when you backspace during an insertion, the characters you backspace over are not erased; the cursor moves backwards, and the characters remain on the display. This may be useful if you are planning to type in something similar to what's already there. In any case, the characters disappear when you hit ESC; if you want to get rid of them immediately, hit an ESC and then a again.

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

3.4.2 Making Small Corrections

To make small corrections in existing text, use the arrow keys, word-length motion commands, backspace (the BACK SPACE key, ↑H, or even just h), or SPACE bar to move the cursor to the incorrect character. If the character is not needed then hit the x key; this deletes the character from the file. It is analogous to the way you x out characters when you make mistakes on a typewriter.

If the character is incorrect, you can replace it with the correct character by giving the command **rc**, where *c* is replaced by the correct character. Finally, if the character which is incorrect should be replaced by more than one character, use the **s** command. This substitutes a string of characters, ending with ESC, for the character under the cursor. If there are a small number of characters that are wrong, precede **s** with a count of the number of characters to be replaced. Counts are also useful with **x** to specify the number of characters to be deleted.

3.4.3 More Corrections: Operators

You already know almost enough to make changes at a higher level. All you need to know now is that the **d** key acts as a delete operator. Try the command **dw** to delete a word. Try hitting **.** (dot) a few times. Notice that this repeats the effect of the **dw**. The command **."** repeats the last command which made a change.

Now try **db**. This deletes a word backwards (it deletes the preceding word). Try **dSPACE**. This deletes a single character, and is equivalent to the **x** command.

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

3.4.4 Operating on Lines

It is often the case that you want to operate on entire lines. Find a line which you want to delete, and type **dd**, the **d** operator twice. This will delete the line. If you are on a dumb terminal, the editor may just erase the line on the screen, replacing it with a line with only an **@** on it. This line does not correspond to any line in your file, but only acts as a place holder. It helps to avoid a lengthy redraw of the rest of the screen which would be necessary to close up the hole created by the deletion on a terminal without a delete line capability.

Try repeating the **c** operator twice; this will change a whole line, erasing its previous contents and replacing them with text you type up to an ESC.

Note: The command **S** is a convenient synonym for **cc**, by analogy with **s**. Think of **S** as a substitute on lines, while **s** is a substitute on characters.

You can delete or change more than one line by preceding the **dd** or **cc** with a count, i.e. **5dd** deletes 5 lines. You can also give a command like **dL** to delete all the lines up to and including the last line on the screen, or **d3L** to delete through the third line from the bottom.

Note: Using the / search after a **d** will normally delete characters from the current position to the point of the match. If you want to delete whole lines including the two points, give the pattern as **/pat/+0**, a line address.

Notice that the editor lets you know when you change a large number of lines so that you can see the extent of the change. The editor will also always tell you when a change you make affects text which you cannot see.

3.4.5 Undo

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

The undo command lets you reverse only a single change. After you make a number of changes to a line, you may decide that you would rather have the original state of the line back. The **U** command restores the current line to the state before you started changing it.

You can recover text which you delete, even if undo will not bring it back; see the section on recovering lost text below.

3.4.6 Summary of Insert/Delete Functions

SPACE	advance the cursor one position
↑H	backspace the cursor
↑W	erase a word during an insert
erase	your erase (usually ↑H or #), erases a character during an insert
kill	your kill (usually @, ↑X, or ↑U), kills the insert on this line
.	repeats the changing command
O	opens and inputs new lines, above the current
U	undoes the changes you made to the current line
a	appends text after the cursor
c	changes the object you specify to the following text
d	deletes the object you specify
i	inserts text before the cursor
o	opens and inputs new lines, below the current
u	undoes the last change

3.5 MOVING, REARRANGING, AND DUPLICATING TEXT

3.5.1 Low Level Character Motions

Move the cursor to a line that includes a parenthesis, comma, or period. Try the command **fx** where **x** is this character. This command finds the next **x** character to the right of the cursor in the current line. Try then hitting a **;**, which finds the next instance of the same character. By using the **f** command and then a sequence of **;**'s you can often get to a

particular place in a line much faster than with a sequence of word motions or SPACES. There is also a **F** command, which is like **f**, but searches backward. The **;** command repeats **F** also.

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

When working with the text of a single line, an **↑** moves the cursor to the first non-white position on the line, and a **\$** moves it to the end of the line. Thus **\$a** will append new text at the end of the current line.

Your file may have tab (**↑I**) characters in it. These characters are represented as a number of spaces expanding to a tab stop, where tab stops are every 8 positions.

Note: This is settable by a command of the form **:se ts=*x***, where *x* is 4 to set tabstops every four columns. This has effect on the screen representation within the editor.

When the cursor is at a tab, it sits on the last of the several spaces which represent that tab. Try moving the cursor back and forth over tabs so you understand how this works.

On rare occasions, your file may include nonprinting characters. These characters are displayed in the same way they are represented in this document, that is with a two character code, the first character of which is "↑". On the screen, non-printing characters resemble one "↑" character adjacent to another, but spacing or backspacing over the character will reveal that the two characters are, like the spaces representing a tab character, a single character.

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

3.5.2 Higher-Level Text Objects

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

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

The operations { and } move over paragraphs and the operations [[and]] move over sections.

Note: The [[and]] operations require the operation character to be doubled because they can move the cursor far from where it currently is. While it is easy to get back with the command ``, these commands would still be frustrating if they were easy to hit accidentally.

A paragraph begins after each empty line, and also at each of a set of paragraph macros, specified by the pairs of characters in the definition of the string valued option **paragraphs**. The default setting for this option defines the paragraph macros of the *-ms* and *-mm* macro packages, i.e. the “.IP”, “.LP”, “.PP” and “.QP”, “.P” and “.LI” macros.

Note: You can easily change or extend this set of macros by assigning a different string to the **paragraphs** option in your EXINIT. The “.bp” **troff** request is also assumed to indicate the start of a paragraph.

Each paragraph boundary is also a sentence boundary. The sentence and paragraph commands, if given counts, can operate over groups of sentences and paragraphs.

Sections in the editor begin after each macro in the *sections* option, normally “.NH”, “.SH”, “.H” and “.HU”, and each line with a formfeed ↑L in the first column. Section boundaries are always line and paragraph boundaries also.

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

3.5.3 Rearranging and Duplicating Text

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

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

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

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

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

3.5.4 Summary of Higher-Level Motions and Objects

↑	first non-white on line
\$	end of line
)	forward sentence
}	forward paragraph
]]	forward section
(backward sentence
{	backward paragraph
[[backward section
<i>fx</i>	find <i>x</i> forward in line
p	put text back, after cursor or below current line
y	yank operator, for copies and moves
<i>tx</i>	up to <i>x</i> forward, for operators
Fx f	backward in line
P	put text back, before cursor or above current line
Tx t	backward in line

3.6 HIGH LEVEL COMMANDS

3.6.1 Writing, Quitting, Editing New Files

So far we have seen how to enter **vi** and to write out our file using either **ZZ** or **:wRETURN**. The first exits from the editor, (writing if changes were made), the second writes and stays in the editor.

If you have changed the editor's copy of the file but haven't written the changes with **:w**, and you don't wish to save the changes, perhaps because you made some major mistakes while editing, or because you decided that the changes did not improve the file, then you can give the command **:q!RETURN** to quit from the editor without writing the changes. You can also re-edit the same file (starting over) by giving the command **:e!RETURN**. Use these commands carefully. It is not possible to recover the changes you have made after you discard them in this manner. As above, re-editing a file with **:e!** only discards changes made since the last **:w** command you gave in that file.

You can edit a different file without leaving **vi** by giving the command **:enameRETURN**. If you have not written out your file before you try to do this, then the editor will tell you this, and delay editing the other file. You can then give the command **:wRETURN** to save your work and then the **:e nameRETURN** command again, or carefully give the command **:e!nameRETURN**, which edits the other file discarding the changes you have made to the current file. To have the editor automatically save changes, include *set autowrite* in your EXINIT file, and use **:n** instead of **:e**.

3.6.2 Escaping to a Shell

You can get to a shell to execute a single command by giving a **vi** command of the form **!:cmdRETURN**. The system will run the single command *cmd* and when the command finishes, the editor will ask you to hit a RETURN to continue. When you have finished looking at the output on the screen, you should hit RETURN and the editor will clear the screen

and redraw it. You can then continue editing. You can also give another `:` command when it asks you for a RETURN; in this case, the screen will not be redrawn.

If you wish to execute more than one command in the shell, then you can give the command `:shRETURN` to get a new shell. When you finish with the shell, ending it by typing a `↑D`, `vi` will clear the screen and continue.

Note: You can not invoke `cs`, the C shell, in this way.

On systems which support it, `↑Z` will suspend the editor and return to the (top level) shell. When the editor is resumed, the screen will be redrawn.

3.6.3 Marking and Returning

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

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

3.6.4 Adjusting the Screen

If the screen image is messed up because of a transmission error to your terminal, or because some program other than the editor wrote output to your terminal, you can hit a `↑L`, the ASCII form-feed character, to refresh the screen.

On a dumb terminal, if there are `@` lines in the middle of the screen as a result of line deletion, you may get rid of them by typing `↑R`. This causes the editor to retype the screen, closing up these holes.

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

3.7 ADVANCED TOPICS

3.7.1 Editing on Slow Terminals

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

The use of the slow terminal insertion mode is controlled by the *slowopen* option. You can force the editor to use this mode even on faster terminals by giving the command `:se slowRETURN`. If your system is sluggish this throttles the amount of output coming to your terminal. You can disable this option by `:se noslowRETURN`.

The editor can simulate an intelligent terminal on a dumb one. Try giving the command `:se redrawRETURN`. This simulation generates a great deal of output and is generally tolerable only on lightly loaded systems and fast terminals. You can disable this by giving the command `:se noredrawRETURN`.

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

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

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

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

You can easily expand or contract the window, placing the current line as you choose, by giving a number on a `z` command, after the `z` and before the following RETURN, . or -. Thus the command `z5.` redraws the screen with the current line in the center of a five line window.

Note: The command `5z.` has an entirely different effect, placing line 5 in the center of a new window.

If the editor is redrawing or otherwise updating large portions of the display, you can interrupt this updating by hitting a `↑I`. If you do this you may partially confuse the editor about what is displayed on the screen. You can still edit the text on the screen if you wish; clear up the confusion by hitting a `↑L`; or move or search again, ignoring the current

state of the display.

3.7.2 Options, Set, and Editor Startup Files

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

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

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

```
set opt=val
```

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

```
set opt
set noopt
```

These statements can be placed in your EXINIT in your environment, or given while you are running **vi** by preceding them with a **:** and following them with a RETURN.

You can get a list of all options which you have changed by the command **:setRETURN**, or the value of a single option by the command **:set opt?RETURN** where *opt* is the option. A list of all possible options and their values is generated by **:set allRETURN**. Set can be abbreviated **se**. Multiple options can be placed on one line, e.g. **:se ai aw nuRETURN**.

Options set by the **set** command only last while you stay in the editor. It is common to want to have certain options set whenever you use the editor. This can be accomplished by creating a list of **ex** commands that are to be run every time you start up **ex**, **edit**, or **vi**.

Note: All commands which start with **:** are **ex** commands.

A typical list includes a **set** command, and possibly a few **map** commands. Since it is advisable to get these commands on one line, they can be separated with the **|** character, for example:

```
set ai aw terse|map @ dd|map # x
```

which sets the options **autoindent**, **autowrite**, **terse**, (the **set** command), makes **@** delete a line, (the first **map**), and makes **#** delete a character, (the second **.B** map). This string should be placed in the variable **EXINIT** in your environment. If you use the C-Shell, put this line in the file *.login* in your home directory:

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

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

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

Of course, the particulars of the line would depend on which options you wanted to set.

3.7.3 Recovering Lost Lines

Vi saves the last 9 deleted blocks of text in a set of registers numbered 1–9. You can get the *n*'th previous deleted text back in your file by the command **"n p**. The **"** here says that a buffer name is to follow, *n* is the number of the buffer you wish to try (use the number 1 for now), and **p** is the put command, which puts text in the buffer after the cursor. If this doesn't bring back the text you wanted, hit **u** to undo this and then **.** (dot) to repeat the put command. In general the **.** command will repeat the last change you made. As a special case, when the last command refers to a numbered text buffer, the **.** command increments the number of the buffer before repeating the command. Thus a sequence of the form

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

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

3.7.4 Recovering Lost Files

If the system crashes, you can recover the work you were doing to within a few changes. You will normally receive mail when you next log in giving you the name of the file which has been saved for you. You should then change to the directory where you were when the system crashed and give a command of the form:

```
% vi -r name
```

replacing *name* with the name of the file which you were editing. This will recover your work to a point near where you left off.

Note: In rare cases, some of the lines of the file may be lost. The editor will give you the numbers of these lines and the text of the lines will be replaced by the string "LOST". These lines will almost always be among the last few which you changed. You can either choose to discard the changes which you made (if they are easy to remake) or to replace the few lost lines by hand.

You can get a listing of the files which are saved for you by giving the command:

```
% vi -r
```

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

For this feature to work, **vi** must be correctly installed by your System Administrator, and the **mail** program must exist to receive mail. The invocation "**vi -r**" will not always list all saved files, but they can be recovered even if they are not listed.

3.7.5 Continuous Text Input

When you are typing in large amounts of text it is convenient to have lines broken near the right margin automatically. You can cause this to happen by giving the command **:se wm=10RETURN**. This causes all lines to be broken at a space at least 10 columns from the right hand edge of the screen.

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

3.7.6 Features for Program Editing

Vi has a number of commands for editing programs. The thing that most distinguishes editing of programs from editing of text is the desirability of maintaining an indented structure to the body of the program. The editor has an **autoindent** facility for helping you generate correctly indented programs.

To enable this facility you can give the command **:se aiRETURN**. Now try opening a new line with **o** and type some characters on the line after a few tabs. If you start another line, notice that the editor supplies white space at the beginning of the line to line it up with the previous line. You cannot backspace over this indentation, but you can use **↑D** key to backtab over the supplied indentation.

Each time you type `↑D` you back up one position, normally to an 8th column boundary. This amount is settable; the editor has an option called `shiftwidth` which lets you change this value. Try giving the command `:se sw=4RETURN` and then experimenting with autoindent again.

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

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

If you are editing C programs, you can use the `[[` and `]]` keys to advance or retreat to a line starting with a `{`, i.e., a function declaration at a time. When `]]` is used with an operator, it stops after a line which starts with `}`; this is sometimes useful with `y]]`.

3.7.7 Filtering Portions of the Buffer

You can run system commands over portions of the buffer using the operator `!`. You can use this to sort lines in the buffer, or to reformat portions of the buffer with a pretty-printer. Try typing in a list of random words, one per line and ending them with a blank line. Back up to the beginning of the list, and then give the command `!}sortRETURN`. This says to sort the next paragraph of material, and the blank line ends a paragraph.

3.7.8 Commands for Editing LISP

If you are editing a LISP program, you should set the option `lisp` by doing

```
:se lispRETURN.
```

This changes the `(` and `)` commands to move backwards and forwards over s-expressions. The `{` and `}` commands are like `(` and `)` but don't stop at atoms. These can be used to skip to the next list, or through a comment quickly.

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

There is another option which is useful for typing in LISP, the `showmatch` option. Try setting it with

```
:se smRETURN
```

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

The editor also has an operator to realign existing lines as though they had been typed in with **lisp** and **autoindent** set. This is the **=** operator. Try the command **=%** at the beginning of a function. This will realign all the lines of the function declaration.

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

3.7.9 Macros

Vi has a parameterless macro facility, which lets you set it up so that when you hit a single keystroke, the editor will act as though you had hit some longer sequence of keys. You can set this up if you find yourself typing the same sequence of commands repeatedly.

Briefly, there are two types of macros:

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

```
:map lhs rhsRETURN
```

mapping *lhs* into *rhs*. There are restrictions: *lhs* should be one keystroke (either 1 character or one function key) since it must be entered within one second (unless *notimeout* is set, in which case you can type it as slowly as you wish, and **vi** will wait for you to finish it before it echoes anything). The *lhs* can be no longer than 10 characters, the *rhs* no longer than 100. To get a space, tab or newline into *lhs* or *rhs* you should escape them with a **↑V**. (It may be necessary to double the **↑V** if the map command is given inside **vi**, rather than in *ex*.) Spaces and tabs inside the *rhs* need not be escaped.

Thus to make the **q** key write and exit the editor, you can give the command

```
:map q :wq↑V↑VRETURN RETURN
```

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

Macros can be deleted with

```
unmap lhs
```


If the *lhs* of a macro is “#0” through “#9”, this maps the particular function key instead of the 2-character “#” sequence. So that terminals without function keys can access such definitions, the form “#x” will mean function key *x* on all terminals (and need not be typed within one second.) The character “#” can be changed by using a macro in the usual way:

```
:map ↑V↑V↑I #
```

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

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

Placing a “!” after the word **map** causes the mapping to apply to input mode, rather than command mode. Thus, to arrange for ↑T to be the same as 4 spaces in input mode, you can type:

```
:map ↑T ↑V♯♯♯♯
```

where ♯ is a blank. The ↑V is necessary to prevent the blanks from being taken as white space between the *lhs* and *rhs*.

3.8 ABBREVIATIONS

3.8.1 Word Abbreviations

Word abbreviation is similar to the macro feature. It allows you to type a short word and have it expanded into a longer word or words. The commands are **:abbreviate** and **:unabbreviate** (**:ab** and **:una**) and have the same syntax as **:map**. For example:

```
:ab dfs distributed file system
```

causes the word “dfs” to always be changed into the phrase “distributed file system”. Word abbreviation is different from macros in that only whole words are affected. If “dfs” were typed as part of a larger word, it would be left alone. Also, the partial word is echoed as it is typed. There is no need for an abbreviation to be a single keystroke, as it should be with a macro.

3.8.2 Editor Command Abbreviations

The editor has a number of short commands that abbreviate longer commands which we have introduced here. They often save a bit of typing, and you can learn them as convenient.

3.9 MORE DETAILS

This section includes information on **vi** commands that will probably be of interest only to those who are doing advanced or specialized editing tasks. This information is not required knowledge for those who are merely using **vi** to edit text.

3.9.1 Line Representation in the Display

The editor folds long logical lines onto many physical lines in the display. Commands which advance lines advance logical lines and will skip over all the segments of a line in one motion. The command `|` moves the cursor to a specific column, and may be useful for getting near the middle of a long line to split it in half. Try `80|` on a line which is more than 80 columns long.

Note: You can make long lines very easily by using `J` to join together short lines.

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

If you wish, you can have the editor place line numbers before each line on the display. Give the command `:se nuRETURN` to enable this, and the command `:se nonuRETURN` to turn it off. You can have tabs represented as `↑I` and the ends of lines indicated with “\$” by giving the command `:se listRETURN`; `:se nolistRETURN` turns this off.

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

3.9.2 Counts

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

new window size	<code>: / ? [[] ` `</code>
scroll amount	<code>↑D ↑U</code>
line/column number	<code>z G </code>
repeat effect	most of the rest

The editor maintains a notion of the current default window size. On terminals which run at speeds greater than 1200 baud, the editor uses the full terminal screen. On terminals which are slower than 1200 baud (most dialup lines are in this group), the editor uses 8 lines as the default window size. At 1200 baud, the default is 16 lines.

This size is the size used when the editor clears and refills the screen after a search or other motion moves far from the edge of the current window. The commands which take a new window size as *count* often cause the screen to be redrawn. If you anticipate this, but do not need as large a window as you are currently using, you may wish to change the screen size by specifying the new size before these commands. In any

case, the number of lines used on the screen will expand if you move off the top with a `-` or similar command or off the bottom with a command such as `RETURN` or `↑D`. The window will revert to the last specified size the next time it is cleared and refilled.

Note: This will not happen if you use a `↑L`, which just redraws the screen as it is.

The scroll commands `↑D` and `↑U` likewise remember the amount of scroll last specified, using half the basic window size initially. The simple insert commands use a count to specify a repetition of the inserted text. Thus `10a+—ESC` will insert a grid-like string of text. A few commands also use a preceding count as a line or column number.

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

3.9.3 More File Manipulation Commands

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

<code>:w</code>	write back changes
<code>:wq</code>	write and quit
<code>:x</code>	write (if necessary) and quit (same as <code>ZZ</code>).
<code>:e</code>	<i>name</i> edit file <i>name</i>
<code>:e!</code>	re-edit, discarding changes
<code>:e</code>	<code>+ name</code> edit, starting at end
<code>:e</code>	<code>+n</code> edit, starting at line <i>n</i>
<code>:e</code>	<code>#</code> edit alternate file
<code>:w</code>	<i>name</i> write file <i>name</i>
<code>:w!</code>	<i>name</i> overwrite file <i>name</i>
<code>:x,yw</code>	<i>name</i> write lines <i>x</i> through <i>y</i> to <i>name</i>
<code>:r</code>	<i>name</i> read file <i>name</i> into buffer
<code>:r</code>	<code>!cmd</code> read output of <i>cmd</i> into buffer
<code>:n</code>	edit next file in argument list
<code>:n!</code>	edit next file, discarding changes to current
<code>:n</code>	<i>args</i> specify new argument list
<code>:ta</code>	<i>tag</i> edit file containing tag <i>tag</i> , at <i>tag</i>

All of these commands are followed by a `RETURN` or `ESC`. The most basic commands are `:w` and `:e`. A normal editing session on a single file will end with a `ZZ` command. If you are editing for a long period of time you can give `:w` commands occasionally after major amounts of editing, and then finish with a `ZZ`. When you edit more than one file, you can finish with one by doing a `:w` then start editing a new file by

giving a **:e** command, or set **autowrite** and use **:n <file>**.

If you make changes to the editor's copy of a file, but do not wish to write them back, then you must give an **!** after the command you would otherwise use; this forces the editor to discard any changes you have made. Use this carefully.

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

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

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

If you wish to edit a set of files in succession, you can give all the names on the command line, and then edit each one in turn using the command **:n**. It is also possible to respecify the list of files to be edited by giving the **:n** command a list of file names, or a pattern to be expanded as you would have given it on the initial **vi** command.

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

3.9.4 More About Searching for Strings

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

thus, use "+0" to affect up to the line which matches.

You can have the editor ignore the case of words in the searches it does by giving the command `:se icRETURN`. The command `:se noicRETURN` turns this off.

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

```
set nomagic
```

in your EXINIT. In this case, only the characters `↑` and `$` are special in patterns. The character `\` is also then special (as it is most everywhere in the system), and may be used to get at the extended pattern matching facility. It is also necessary to use a `\` before a `/` in a forward scan or a `?` in a backward scan, in any case. The following table gives the extended forms when **magic** is set.

<code>↑</code>	at beginning of pattern, matches beginning of line
<code>\$</code>	at end of pattern, matches end of line
<code>.</code>	matches any character
<code>\<</code>	matches the beginning of a word
<code>\></code>	matches the end of a word
<code>[str]</code>	matches any single character in <i>str</i>
<code>[↑str]</code>	matches any single character not in <i>str</i>
<code>[x-y]</code>	matches any character between <i>x</i> and <i>y</i>
<code>*</code>	matches any number of the preceding pattern

If you use **nomagic** mode, then the `.` [`]` and `*` primitives are given with a preceding `\`.

3.9.5 More About Input Mode

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

<code>↑H</code>	deletes the last input character
<code>↑W</code>	deletes the last input word, defined as by <code>b</code>
<code>erase</code>	your erase character, same as <code>↑H</code>
<code>kill</code>	your kill character, deletes the input on this line
<code>\</code>	escapes a following <code>↑H</code> and your erase and kill
<code>ESC</code>	ends an insertion
<code>DEL</code>	interrupts an insertion, terminating it abnormally
<code>RETURN</code>	starts a new line
<code>↑D</code>	backtabs over <i>autoindent</i>
<code>0↑D</code>	kills all the <i>autoindent</i>
<code>↑↑D</code>	same as <code>0↑D</code> , but restores indent next line
<code>↑V</code>	quotes the next non-printing character into the file

The most common way of making corrections to input is by typing `↑H` to correct a single character, or by typing one or more `↑W`'s to back over incorrect words. If you use `#` as your erase character in the normal system, it will work like `↑H`.

Your system kill character, normally @, ↑X or ↑U, will erase all the input you have given on the current line. In general, you can neither erase input back around a line boundary nor can you erase characters which you did not insert with this insertion command. To make corrections on the previous line after a new line has been started you can hit ESC to end the insertion, move over and make the correction, and then return to where you were to continue. The command A which appends at the end of the current line is often useful for continuing.

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

Note: Almost any character. The implementation of the editor does not allow the NULL (↑@) character to appear in files. Also the LF (linefeed or ↑J) character is used by the editor to separate lines in the file, so it cannot appear in the middle of a line. You can insert any other character, however, if you wait for the editor to echo the ↑ before you type the character. In fact, the editor will treat a following letter as a request for the corresponding control character. This is the only way to type ↑S or ↑Q, since the system normally uses them to suspend and resume output and never gives them to the editor to process.

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

When you are using **autoindent**, you may wish to place a label at the left margin of a line. The way to do this easily is to type ↑ and then ↑D. The editor will move the cursor to the left margin for one line, and restore the previous indent on the next. You can also type a 0 followed immediately by a ↑D if you wish to kill all the indent and not have it come back on the next line.

3.9.6 Uppercase Only Terminals

Note: We do not support uppercase-only terminals.

3.9.7 Vi and ex

Vi is actually one mode of editing within the editor **ex**. When you are running **vi** you can escape to the line-oriented editor of **ex** by giving the

command **Q**. All of the **:** commands which were introduced above are available in **ex**. Likewise, most **ex** commands can be invoked from **vi** using **:**. Just give them without the **:** and follow them with a RETURN.

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

There are a number of things which you can do more easily in **ex** than in **vi**. Systematic changes in line-oriented material are especially easy. You can read the advanced editing documents for the editor **ed** to find out a lot more about this style of editing. Experienced users often mix their use of **ex** command mode and **vi** command mode to speed the work they are doing.

3.9.8 Open Mode: **vi** on Hardcopy Terminals and "Glass TTY's"

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

The only difference between *visual* mode and *open* mode is the way in which the text is displayed.

In *open* mode, the editor uses a single line window into the file, and moving backwards and forwards in the file causes new lines to be displayed, always below the current line. Two **vi** commands work differently in *open* mode: **z** and **↑R**. The **z** command does not take parameters in *open* mode. Instead, it draws a "window of context" around the current line, then returns you to the current line.

If you are using a hardcopy terminal, the **↑R** command retypes the current line as **two** lines: the first line is the unedited line, the second is the edited line. When you delete characters, the editor types a number of ****'s to show you the characters which are deleted. The editor also reprints the current line soon after such changes so that you can see what the line looks like again.

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

3.10 A SUMMARY OF **vi** COMMANDS

This section summarizes the various **vi** editing and cursor motion commands. In it, we use the following notational conventions. **[option]** is used to denote optional parts of a command. Many **vi** commands have

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

{**variable item**} is used to denote parts of the command which must appear, but can take a number of different values.

<**character** [-**character**]> means that the character or one of the characters in the range described between the two angle brackets is to be typed. For example <**esc**> means the **escape** key is to be typed. <**a-z**> means that a lower case letter is to be typed.

↑<**character**> means that the character is to be typed as a **control** character, that is, with the CTRL key held down while simultaneously typing the specified character. In this document, control characters will be denoted using the *uppercase* character, but ↑<uppercase chr> and ↑<lowercase chr> are equivalent. For example, <↑**D**> is equal to <↑**d**>. The most common character abbreviations used in this list are as follows:

<**esc**> escape, octal 033
 <**cr**> carriage return, ↑M, octal 015
 <**lf**> linefeed ↑J, octal 012
 <**nl**> newline, ↑J, octal 012 (same as linefeed)
 <**bs**> backspace, ↑H, octal 010
 <**tab**> tab, ↑I, octal 011
 <**bell**> bell, ↑G, octal 07
 <**ff**> formfeed, ↑L, octal 014
 <**sp**> space, octal 040
 <**del**> delete, octal 0177

3.10.1 Entry and Exit

To enter **vi** on a particular *file*, type

vi file

The file will be read in and the cursor will be placed at the beginning of the first line. The first screenfull of the file will be displayed on the terminal.

To get out of the editor, type

ZZ

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

3.10.2 Cursor and Page Motion

[cnt]<bs> or [cnt]h or [cnt]←

Move the cursor to the left one character. Cursor stops at the left margin of the page. If cnt is given, these commands move that many spaces.

[cnt]↑N or [cnt]j or [cnt]↓ or [cnt]<lf>

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

[cnt]↑P or [cnt]k or [cnt]↑ Move up one line. Moving off the top of the screen forces new text onto the screen. Mnemonic: **P**revious

[cnt]<sp> or [cnt]l or [cnt]→

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

[cnt]-

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

[cnt]+ or [cnt]<cr>

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

[cnt]\$

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

↑

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

0

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

[cnt]|

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

[cnt]w

Move the cursor to the beginning of the next word. If there is a count, then move forward that many words and position the cursor at the beginning of the word. Mnemonic: **n**ext-**w**ord

[cnt]W

Move the cursor to the beginning of the next word which follows a "white space" (<sp>, <tab>, or <nl>). Ignore other punctuation.

- [cnt]b Move the cursor to the preceding word.
Mnemonic: **backup-word**
- [cnt]B Move the cursor to the preceding word that is separated from the current word by a "white space" (<sp>, <tab>, or <nl>).
- [cnt]e Move the cursor to the end of the current word or the end of the "cnt"th word hence.
Mnemonic: **end-of-word**
- [cnt]E Move the cursor to the end of the current word which is delimited by "white space" (<sp>, <tab>, or <nl>).
- [line number]G Move the cursor to the line specified. Of particular use are the sequences "1G" and "G", which move the cursor to the beginning and the end of the file respectively. Mnemonic: **Go-to**

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

- [cnt]↑D Move the cursor down in the file by "cnt" lines (or the last "cnt" if a new count isn't given. The initial default is half a page.) The screen is simultaneously scrolled up. Mnemonic: **Down**
- [cnt]↑U Move the cursor up in the file by "cnt" lines. The screen is simultaneously scrolled down. Mnemonic: **Up**
- [cnt]↑F Move the cursor to the next page. A count moves that many pages. Two lines of the previous page are kept on the screen for continuity if possible.
Mnemonic: **Forward-a-page**
- [cnt]↑B Move the cursor to the previous page. Two lines of the current page are kept if possible. Mnemonic: **Backup-a-page**
- [cnt](Move the cursor to the beginning of the next sentence. A sentence is defined as ending with a ".", "!", or "?" followed by two spaces or a <nl>.
- [cnt]) Move the cursor backwards to the beginning of a sentence.
- [cnt]} Move the cursor to the beginning of the next paragraph. This command works best inside **nroff** documents. It understands two sets of **nroff** macros,

- ms** and **-mm**, for which the commands **"LI"**, **"LP"**, **"PP"**, **"QP"**, **"P"**, as well as the **nroff** command **".bp"**, are considered to be paragraph delimiters. A blank line also delimits a paragraph. The **nroff** macros that it accepts as paragraph delimiters is adjustable. See **paragraphs** under the **Set Commands** section.
- [cnt]{** Move the cursor backwards to the beginning of a paragraph.
-]]** Move the cursor to the next "section", where a section is defined by two sets of **nroff** macros, **-ms** and **-mm**, in which **".H"**, **".SH"**, and **".H"** delimit a section. A line beginning with a **<ff><nl>** sequence, or a line beginning with a **"{"** are also considered to be section delimiters. The last option makes it useful for finding the beginnings of C functions. The **nroff** macros that are used for section delimiters can be adjusted. See **sections** under the **Set Commands** section.
- [[** Move the cursor backwards to the beginning of a section.
- %** Move the cursor to the matching parenthesis or brace. This is very useful in C or lisp code. If the cursor is sitting on a **() { or }**, the cursor is moved to the matching character at the other end of the section. If the cursor is not sitting on a brace or a parenthesis, **vi** searches forward until it finds one and then jumps to the match mate.
- [cnt]H** If there is no count, move the cursor to the top left position on the screen. If there is a count, then move the cursor to the beginning of the line "cnt" lines from the top of the screen. Mnemonic: **Home**
- [cnt]L** If there is no count, move the cursor to the beginning of the last line on the screen. If there is a count, then move the cursor to the beginning of the line "cnt" lines from the bottom of the screen. Mnemonic: **Last**
- M** Move the cursor to the beginning of the middle line on the screen. Mnemonic: **Middle**
- m<a-z>** This command does not move the cursor, but it **marks** the place in the file and the character "**<a-z>**" becomes the label for referring to this location in the file. See the next two commands. Mnemonic: **mark**

Note: The mark command is not a motion. It cannot be used as the target of commands such as delete.

- ' <a-z> Move the cursor to the beginning of the line that is marked with the label " <a-z>".
- ` <a-z> Move the cursor to the exact position on the line that was marked with the label "<a-z>".
- `` Move the cursor back to the beginning of the line where it was before the last "non-relative" move. A "non-relative" move is something such as a search or a jump to a specific line in the file, rather than moving the cursor or scrolling the screen.
- `` Move the cursor back to the exact spot on the line where it was located before the last "non-relative" move.

3.10.3 Searches

The following commands allow you to search for items in a file.

- [cnt]f{chr} Search forward on the line for the next or *cnt*th occurrence of the character *chr*. The cursor is placed **at** the character of interest. Mnemonic: **f**ind character
- [cnt]F{chr} Search backwards on the line for the next or *cnt*'th occurrence of the character "chr". The cursor is placed at the character of interest.
- [cnt]t{chr} Search forward on the line for the next or *cnt*'th occurrence of the character "chr". The cursor is placed just preceding the character of interest. Mnemonic: **m**ove cursor **u**p **t**o character
- [cnt]T{chr} Search backwards on the line for the next or *cnt*'th occurrence of the character "chr". The cursor is placed just preceding the character of interest.
- [cnt]; Repeat the last "f", "F", "t" or "T" command.
- [cnt], Repeat the last "f", "F", "t" or "T" command, but in the opposite search direction. This is useful if you overshoot.
- [cnt]/[string]/RETURN Search forward for the next occurrence of "string". Wraparound at the end of the file does occur. The final </> is not required.

[cnt]?[string]?RETURN

Search backwards for the next occurrence of "string". If a count is specified, the count becomes the new window size. Wraparound at the beginning of the file does occur. The final <?> is not required.

n Repeat the last /[string]/ or ?[string]? search.
Mnemonic: **n**ext occurrence.

N Repeat the last /[string]/ or ?[string]? search, but in the reverse direction.

:g/[string]/[editor command]<nl>

Using the : syntax, it is possible to do global searches in the style of the "ed" editor.

3.10.4 Text Insertion

The following commands allow for the insertion of text. All multicharacter text insertions are terminated with an <esc> character. The last change can always be **undone** by typing a **u**. The text insert in insertion mode can contain newlines.

a{text}<esc> Insert text immediately following the cursor position.
Mnemonic: **a**ppend

A{text}<esc> Insert text at the end of the current line. Mnemonic: **A**ppend

i{text}<esc> Insert text immediately preceding the cursor position. Mnemonic: **i**nsert

I{text}<esc> Insert text at the beginning of the current line.

o{text}<esc> Insert a new line after the line on which the cursor appears and insert text there. Mnemonic: **o**pen new line

O{text}<esc> Insert a new line preceding the line on which the cursor appears and insert text there.

3.10.5 Text Deletion

The following commands allow the user to delete text in various ways. All changes can always be **undone** by typing the **u** command.

[cnt]x Delete the character or characters starting at the cursor position.

[cnt]X Delete the character or characters starting at the character preceding the cursor position.

D Deletes the remainder of the line starting at the cursor. Mnemonic: **D**elete the rest of line

`[cnt]d{motion}` Deletes one or more occurrences of the specified motion. Any motion from sections 4.1 and 4.2 can be used here. The `d` can be stuttered (e.g. `[cnt]dd`) to delete `cnt` lines.

3.10.6 Text Replacement

The following commands let you simultaneously delete and insert new text. All such actions can be **undone** by typing `u` following the command.

`r<chr>` Replaces the character at the current cursor position with `<chr>`. This is a one character replacement. No `<esc>` is required for termination. Mnemonic: replace character

`R{text}<esc>` Starts overlaying the characters on the screen with whatever you type. It does not stop until an `<esc>` is typed.

`[cnt]s{text}<esc>` Substitute for "cnt" characters beginning at the current cursor position. A "\$" will appear at the position in the text where the "cnt"'th character appears so you will know how much you are erasing. Mnemonic: substitute

`[cnt]S{text}<esc>` Substitute for the entire current line (or lines). If no count is given, a "\$" appears at the end of the current line. If a count of more than 1 is given, all the lines to be replaced are deleted before the insertion begins.

`[cnt]c{motion}{text}<esc>` Change the specified "motion" by replacing it with the insertion text. A "\$" will appear at the end of the last item that is being deleted unless the deletion involves whole lines. The specified `{motion}` can be any motion listed in the sections above. Stuttering the `c` (e.g. `[cnt]cc`) changes `cnt` lines.

3.10.7 Moving Text

`Vi` provides a number of ways of moving chunks of text around. There are nine buffers into which each piece of deleted or "yanked" text is put, in addition to the "undo" buffer. The most recent deletion or yank is in the "undo" buffer and also in buffer 1. The next most recent is in buffer 2, and so forth. Each new deletion pushes down all the older deletions. Deletions older than 9 disappear. There is also a set of named registers, `a-z`, into which text can optionally be placed. If any delete or replacement type command is preceded by "`<a-z>`", that named buffer will contain the text deleted after the command is executed. For example, "`a3dd`" will delete three lines starting at the current line and put them in buffer "a".

Note: Referring to an upper case letter as a buffer name (A-Z) is the same as referring to the lower case letter, except that text placed in such a buffer is appended to it instead of replacing it.

There are two more basic commands and some variations useful in getting and putting text into a file.

[<a-z>][cnt]y{motion} Yank the specified item or "cnt" items and put in the "undo" buffer or the specified buffer. The variety of "items" that can be yanked is the same as those that can be deleted with the "d" command or changed with the "c" command. In the same way that "dd" means delete the current line and "cc" means replace the current line, "yy" means yank the current line.

[<a-z>][cnt]Y Yank the current line or the "cnt" lines starting from the current line. If no buffer is specified, they will go into the "undo" buffer, in the same manner as any delete. It is equivalent to "yy". Mnemonic: **Y**ank

[<a-z>]p Put "undo" buffer or the specified buffer down **after** the cursor. If whole lines were yanked or deleted into the buffer, then they will be put down on the line following the line the cursor is on. If something else was deleted, like a word or sentence, then it will be inserted immediately following the cursor. Mnemonic: **p**ut buffer

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

[<a-z>]P Put "undo" buffer or the specified buffer down **before** the cursor. If whole lines were yanked or deleted into the buffer, then they will be put down on the line preceding the line the cursor is on. If something else was deleted, like a word or sentence, then it will be inserted immediately preceding the cursor.

[cnt]>{motion} The shift operator will right shift all the text from the line on which the cursor is located to the line where the **motion** is located. The text is shifted by one **shiftwidth**. >> means right shift the current

line or lines.

- [cnt]<{motion} The shift operator will left shift all the text from the line on which the cursor is located to the line where the **item** is located. The text is shifted by one **shiftwidth**. << means left shift the current line or lines. Once the line has reached the left margin it is not further affected.
- [cnt]={motion} Prettyprints the indicated area according to **lisp** conventions. The area should be a lisp s-expression.

3.10.8 Miscellaneous Commands

- ZZ** This is the normal way to exit from **vi**. If any changes have been made, the file is written out. You are returned to the shell at that point.
- ↑L** Redraw the current screen.
- ↑R** On dumb terminals, those not having the "delete line" function (the vt100 is such a terminal), **vi** saves redrawing the screen when you delete a line by just marking the line with an "@" at the beginning and blanking the line. If you want to actually get rid of the lines marked with "@" and see what the page looks like, typing a ↑R will do this.
- .** "Dot" is a particularly useful command. It repeats the last text modifying command. Therefore, you can type a command once and then move to another place and repeat it by just typing ".".
- u** Perhaps the most important command in the editor, **u** undoes the last command that changed the buffer. Mnemonic: **undo**
- U** Undo all the text modifying commands performed on the current line since the last time you moved onto it.
- [cnt]**J** Join the current line and the following line. The <nl> is deleted and the two lines joined, usually with a space between the end of the first line and the beginning of what was the second line. If the first line ended with a "period", then two spaces are inserted. A count joins the next cnt lines. Mnemonic: **Join lines**
- Q** Switch to **ex** editing mode. In this mode, **vi** will behave very much like **ed**. The editor in this mode will operate on single lines normally and will not attempt to keep the "window" up to date. Once in this mode it is also possible to switch to the **open**

mode of editing. By entering the command [**line number**]**open**<nl>, you enter this mode. It is similar to the normal visual mode except the window is only **one** line long. Mnemonic: **Q**uit visual mode

↑]

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

[cnt]!{motion}{UNIX cmd}<nl>

This **vi** command lets you send a section through any UNIX filter program, then replaces that section of text with the output of that program. Useful examples are programs like **cb**, **sort**, and **nroff**. For instance, using **sort** it would be possible to sort a section of the current file into a new list. Using **!!** means take a line or lines starting at the line the cursor is currently on and pass them to the UNIX command.

z{cnt}<nl>

This resets the current window size to *cnt* lines and redraws the screen.

3.10.9 Special Insert Characters

↑V

During inserts, typing a ↑V allows you to quote control characters into the file. Any character typed after the ↑V will be inserted into the file.

[↑]↑D or [0]↑D

<↑D> without any argument backs up one **shiftwidth**. This is necessary to remove indentation that was inserted by the **autoindent** feature. ↑<↑D> temporarily removes all the autoindentation, thus placing the cursor at the left margin. On the next line, the previous indent level will be restored. This is useful for putting "labels" at the left margin. 0<↑D> says remove all autoindents and stay that way. Thus the cursor moves to the left margin and stays there on successive lines until <tab>'s are typed. As with the <tab>, the <↑D> is only effective before any other "non-autoindent" controlling characters are typed. Mnemonic: **D**elete a shiftwidth

↑W

If the cursor is sitting on a word, <↑W> moves the cursor back to the beginning of the word, thus erasing the word from the insert. Mnemonic: **e**rase **W**ord

<bs> The backspace always serves as an erase during insert modes in addition to your normal "erase" character. To insert a <bs> into your file, use the <↑V> to quote it.

3.10.10 ":" Commands

Typing a ":" during command mode causes **vi** to put the cursor at the bottom on the screen in preparation for a command. In the ":" mode, **vi** can be given most **ed** commands. From this mode, you may exit from **vi** or switch to editing a different file. All commands of this variety are terminated by a <cr> or an <esc>.

- :w[!] [file] Causes **vi** to write out the current text to the disk. It is written to the file you are editing unless "file" is supplied. If "file" is supplied, the write is directed to that file instead. If that file already exists, **vi** will not perform the write unless the "!" is supplied indicating you *really* want to destroy the existing file.
- :q[!] Causes **vi** to exit. If you have modified the file you are looking at currently and haven't written it out, **vi** will refuse to exit unless the "!" is supplied.
- :e[!] [+ [cmd]] [file] Start editing a new file called "file" or start editing the current file over again. The command ":e!" says "ignore the changes I've made to this file and start over from the beginning". It is useful if you make major editing errors. The optional "+" says instead of starting at the beginning, start at the "end", or, if "cmd" is supplied, execute "cmd" first. Useful cases of this are where cmd is "n" (any integer) which starts at line number n, and "/text", which searches for "text" and starts at the line where it is found.
- ↑↑ Switch back to the place you were before your last tag command. If your last tag command stayed within the file, ↑↑ returns to that tag. If you have no recent tag command, it will return to the same place in the previous file that it was showing when you switched to the current file.
- :n[!] Start editing the next file in the argument list. Since **vi** can be called with multiple file names, the ":n" command tells it to stop work on the current file and switch to the next file. If the current file was modified, it has to be written out before the ":n" will work or else the "!" must be supplied, which says discard the changes I made to the current file.
- :n[!] file [file file ...] Replace the current argument list with a new list of files and start editing the first file in this new list.

:r file Read in a copy of "file" on the line after the cursor.

:r !cmd Execute the "cmd" and take its output and put it into the file after the current line.

!:cmd Execute any UNIX shell command.

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

tag filename **vi**-search-command

If **vi** finds the tag you specified in the **:ta** command, it stops editing the current file if necessary and if the current file is up to date on the disk, it switches to the file specified and uses the search pattern specified to find the "tagged" item of interest. This is particularly useful when editing multifile C programs. There is a program called **ctags** which generates an appropriate **tags** file for C and FORTRAN programs so that by saying **:ta function <nl>** you will be switched to that function. It could also be useful when editing multifile documents, though the **tags** file would have to be generated manually.

3.10.11 Special Arrangements for Startup

Vi takes the value of **\$TERM** and looks up the characteristics of that terminal in the file **/etc/termcap**. If you don't know **vi**'s name for the terminal you are working on, look in **/etc/termcap**.

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

To recover from a crash or inadvertent hangup, re-establish contact with a UNIX shell, then type:

vi -r file

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

3.10.12 Set Commands

Vi has a number of internal variables and switches which can be set to achieve special effects. These options come in three forms, those that are switches, which toggle from off to on and back, those that require a numeric value, and those that require an alphanumeric string value. The

toggle options are set by a command of the form:

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

and turned off with the command:

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

Commands requiring a value are set with a command of the form:

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

To display the value of a specific option type:

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

To display only those that you have changed type:

```
:set<nl>
```

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

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

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

To arrange to have values other than the default used every time you enter **vi**, place the appropriate **set** command in **EXINIT** in your environment, e.g.

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

or

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

for the Bourne and C Shells respectively. These are usually placed in your **.profile** or **.login**.

autoindent ai	Default: noai Type: toggle When in autoindent mode, vi helps you indent code by starting each line in the same column as the preceding line. Tabbing to the right with <tab> or <↑T> will move this boundary to the right. It can be moved to the left with <↑D>.
autoprint ap	Default: ap Type: toggle Causes the current line to be printed after each ex text modifying command. This is not of much interest in the normal vi visual mode.
autowrite aw	Default: noaw type: toggle Autowrite causes an automatic write to be done if there are unsaved changes before certain commands which change files or otherwise interact with the

- outside world. These commands are `!:`, `:tag`, `:next`, `:rewind`, `↑↑`, and `↑↓`.
- beautify bf** Default: nobf Type: toggle
Causes all control characters except `<tab>`, `<nl>`, and `<ff>` to be discarded.
- directory dir** Default: dir=/tmp Type: string
This is the directory in which **vi** puts its temporary file.
- errorbells eb** Default: noeb Type: toggle
Error messages are preceded by a `<bell>`.
- hardtabs ht** Default: hardtabs=8 Type: numeric
This option contains the value of hardware tabs in your terminal, or of software tabs expanded by `DOMAIN/IX`.
- ignorecase ic** Default: noic Type: toggle
All upper case characters are mapped to lower case in regular expression matching.
- lisp** Default: nolisp Type: toggle
Autoindent for **lisp** code. The commands `()` `[[` and `]]` are modified appropriately to affect s-expressions and functions.
- list** Default: nolist Type: toggle
All printed lines have the `<tab>` and `<nl>` characters displayed visually.
- magic** Default: magic Type: toggle
Enable the metacharacters for matching. These include `.` `*` `<` `>` `[string]` `[↑string]` and `[<chr>-<chr>]`.
- number nu** Default: nonu Type: toggle.
Each line is displayed with its line number.
- open** Default: open Type: toggle
When set, prevents entering open or visual modes from **ex** or **edit**.
- optimize opt** Default: opt Type: toggle
Basically of use only when using the **ex** capabilities. This option prevents automatic `<cr>`s from taking place, and speeds up output of indented lines.
- paragraphs para** Default: para=IPLPPPQPP bp Type: string
Each pair of characters in the string indicate **nroff** macros which are to be treated as the beginning of a paragraph for the `{` and `}` commands. The default string is for the **-ms** and **-mm** macros. To indicate

one-letter **nroff** macros, such as **.P** or **.H**, quote a space in for the second character position. For example:

```
:set paragraphs=P\ bp<nl>
```

would cause **vi** to consider **.P** and **.bp** as paragraph delimiters.

- prompt** Default: prompt Type: toggle
In **ex** command mode the prompt character **:** will be printed when **ex** is waiting for a command. This is not of interest from **vi**.
- redraw** Default: noredraw Type: toggle
On dumb terminals, force the screen to always be up to date, by sending great amounts of output. Useful only at high speeds.
- report** Default: report=5 Type: numeric
This sets the threshold for the number of lines modified. When more than this number of lines are modified, removed, or yanked, **vi** will report the number of lines changed at the bottom of the screen.
- scroll** Default: scroll={1/2 window} Type: numeric
This is the number of lines that the screen scrolls up or down when using the **<↑U>** and **<↑D>** commands.
- sections** Default: sections=SHNHH HU Type: string
Each two-character pair of this string specify **nroff** macro names which are to be treated as the beginning of a section by the **]]** and **[[** commands. The default string is for the **-ms** and **-mm** macros. To enter one-letter **nroff** macros, use a quoted space as the second character. See **paragraphs** for a fuller explanation.
- shell sh** Default: sh=from environment SHELL or /bin/sh
Type: string
This is the name of the **sh** to be used for "escaped" commands.
- shiftwidth sw** Default: sw=8 Type: numeric
This is the number of spaces that a **<↑T>** or **<↑D>** will move over for indenting, and the amount **<** and **>** shift by.
- showmatch sm** Default: nosm Type: toggle
When a **)** or **}** is typed, show the matching **(** or **{** by moving the cursor to it for one second if it is on the current screen.

- slowopen slow** Default: terminal dependent Type: toggle
On terminals that are slow and unintelligent, this option prevents the updating of the screen some of the time to improve speed.
- tabstop ts** Default: ts=8 Type: numeric
<tab>s are expanded to boundaries that are multiples of this value.
- taglength tl** Default: tl=0 Type: numeric
If nonzero, tag names are only significant to this many characters.
- term** Default: (from environment **TERM**, else dumb)
Type: string
This is the terminal and controls the visual displays. To change **term** when in "visual" mode, you must Q to command mode, type a set term command, then re-enter **vi**. (You may also exit **vi**, change \$TERM, and reenter.) The definitions that drive a particular terminal type are found in the file **/etc/termcap**.
- terse** Default: terse Type: toggle
When set, the error diagnostics are short.
- warn** Default: warn Type: toggle
You are warned if you try to escape to the shell without writing out the current changes.
- window** Default: window={8 at 600 baud or less, 16 at 1200 baud, and screen size - 1 at 2400 baud or more}
Type: numeric
This is the number of lines in the window whenever **vi** must redraw an entire screen. It is useful to make this size smaller if you are on a slow line.
- w300, w1200, w9600** These set window, but only within the corresponding speed ranges. They are useful in an EXINIT to fine-tune window sizes. For example,
- set w300=4 w1200=12
- causes a 4 lines window at speed up to 600 baud, a 12 line window at 1200 baud, and a full screen (the default) at over 1200 baud.
- wrapscan ws** Default: ws Type: toggle
Searches will wraparound the end of the file when this option is set. When it is off, the search will terminate when it reaches the end or the beginning of the file.
- wrapmargin wm** Default: wm=0 Type: numeric
Vi will automatically insert a <nl> when it finds a

natural break point (usually a <sp> between words) that occurs within "wm" spaces of the right margin. Therefore with "wm=0" the option is off. Setting it to 10 would mean that any time you are within 10 spaces of the right margin **vi** would be looking for a <sp> or <tab> which it could replace with a <nl>. This is convenient for people who forget to look at the screen while they type.

writeany wa

Default: nowa Type: toggle

Vi normally makes a number of checks before it writes out a file. This prevents the user from inadvertently destroying a file. When the "writeany" option is enabled, **vi** no longer makes these checks.

Chapter 4: An Introduction to the DM Editor

4.1 THE DISPLAY MANAGER EDITOR

The Display Manager (DM) is the program that controls the display screen of a DOMAIN node. In addition to its window-management functions, the DM includes a highly programmable full-screen editor. This editor handles all manipulation of text on the screen of your DOMAIN node. It allows you to

- edit commands in the input pad of a shell window
- edit text in an “edit pad” window
- search the contents of an edit or transcript pad for a particular pattern of characters
- copy text from one window and paste it into another window (or another place in the same window) or write it to a disk file
- redefine the keyboard and function keys to suit the needs of the task at hand.

In this chapter, we provide an introduction to the DM editor for DOMAIN/IX users. Even though DOMAIN/IX includes several popular UNIX editors (**ed**, **ex**, **vi**), the DM editor offers something the others don't: a uniform editorial interface between you and any process requiring keyboard input.

All of the editing features described in this chapter apply to shell input pads as well as to the edit pads used for creating text files. In addition, the DM editor's pattern matching facilities can be used to search through shell transcripts for data, old command lines, error messages, filenames, and similar things. And, while you can't edit a transcript pad, you can cut material out of it and paste it back into an input pad, where it can be edited or resubmitted as is. You can also save sections of shell transcripts for later examination and analysis. (See Chapter 1 of the *DOMAIN/IX User's Guide* for more on shells and transcript pads.) You may prefer another editor for certain specific tasks, but a short time spent learning the fundamentals of the DM editorial interface will allow you to use the DOMAIN system in the most pleasant and efficient manner.

Each section in this chapter describes a set of editing tasks and the DM commands you use to perform them. You can execute a DM command by:

- pressing a key that has been mapped to a particular DM command or command sequence (either with the default key definitions, or by a

key definitions file you create),

- entering the command(s) at the

Command:

prompt in the DM input window.

The information in this chapter is only an introduction, meant to give you some indication of what the DM editor does and — in some cases — how it does it. For a complete description of all the DM editing commands described in this chapter, refer to the *DOMAIN System Command Reference*.

- : When you create a file using the DM editor, UNIX programs will see it as owned by “root” until you explicitly specify another owner of the file using the **chown**[1] command. In this case, ownership is assigned to “root” only because the real owner can’t be determined. You will not have to log in as “root” in order to change the ownership of these files. Once ownership has been assigned, it will not be affected by further editing with the DM editor. It is especially important to recognize this phenomenon when using the DM editor to create *.login*, *.cshrc* and *.profile* files, since UNIX shells only read these files if they are owned by the person opening the shell.

4.2 OPENING AN EDIT PAD

To open an edit pad, use the DM command **ce** (normally mapped to the **[EDIT]** key). If you press the **[EDIT]** key, you will be prompted to type a filename

edit file:

in the DM input window. You may also use the direct form of the **ce** command

Command: **ce filename**

where *filename* is the name of the file you want to edit.

If the named file exists, the DM will open an edit pad onto it and place the cursor over the first character in the file. If the named file does not exist, the DM will create it and open a blank edit pad.

If you want to open the pad in read-only mode, use the DM command **cv** (normally mapped to the **[READ]** key). If you press the **[READ]** key, you will be prompted

read file:

in the DM input window. You may also use the direct form of the **cv** command

Command: **cv** *filename*

where *filename* is the name of the file you want to edit.

If the named file exists, the DM will open an edit pad onto it, place the pad in read-only mode, and place the cursor over the first character in the file. If the named file does not exist, the DM will return an error message.

4.3 SAVING THE CONTENTS OF AN EDIT PAD

An edit pad is a volatile area. All editing is done in a buffer, so all changes made in an edit pad must be explicitly written to the file, otherwise they will be lost. That's why the DM's **pw** (pad write) command is so important. The **pw** command is normally mapped to the **SAVE** key. You can also execute it in the DM input window by typing

Command: **pw**

If you follow a **pw** with a **wc** command, the contents of the pad will be written to disk and the window onto the pad will be closed. Otherwise, the file will simply be updated and the pad left open.

Note: If you close a window onto an edit pad (by executing a **wc** command), all changes made since the last **pw** will be lost. If you attempt to do this, the DM will prompt you with the following message in the DM input window.

File modified. OK to quit?

You must type either **y** or **n**. An **n** will return the cursor to the edit pad. A **y** will close the file and discard the changes.

The sequence **pw; wc** (pad write, window close) is usually mapped to ↑Y. Pressing **CTRL** **Y** will update the file and close the window. The command **pw** is usually mapped to ↑W, so pressing **CTRL** **W** will update the file and leave the window open.

The first time you execute a **pw** during an editing session on the file *filename*, the DM, as you would expect, writes the contents of the edit pad to the disk file *filename*. The next time you do a **pw** on *filename*, the previous version is renamed *filename.bak*. In this way, the *dm* always keeps two versions of any file that has been saved more than once. The *filename* version, which is the most recent, and the *filename.bak* version, which is the second most recent.

4.4 EDIT PAD MODES

Edit pads can be opened in read-only mode or read/write mode. You cannot, of course, make changes to the text in a read-only edit pad, although you can copy, search, and scroll through the text. In write mode, you can write to a pad and change text using all of the editing

commands described in this chapter.

When a pad is in read-only mode, the letter **R** appears in the window legend. The R disappears when the pad is put into read/write mode. The DM command **ro** (normally mapped to ↑M) sets read-only mode. It has the following format.

Command: **ro** [-on|-off]

If you do not specify an option, **ro** toggles the current mode setting. If you've modified the text in a pad, you cannot change the pad to read-only mode without first writing the changes to a disk file (saving the file). The **pw** command, described in the previous section, allows you to write your changes to a disk file without closing the pad and window.

The DM editor defaults to insert mode, although it can be reset to overstrike mode. In insert mode, the DM inserts characters you type at the current cursor position. The remainder of the line moves right to make room for the new characters. In overstrike mode, characters you type replace those under the cursor.

When a pad is in insert mode, the letter **I** appears in the window legend. The I disappears when the pad is put into overstrike mode. All read/write pads are initially opened in insert mode.

You can toggle in and out of insert mode by using the **INS** key (called **INS MODE** on 880 keyboards). You can also use the DM command **ei**, which has the following format:

Command: **ei** [-on|-off]

If you do not specify an option, the **ei** toggles the current mode.

Any attempt to type past the window border will result in a beep and a

No room for more text at that position

message from the DM. When this happens, press **RETURN** at the beginning of the next line.

4.5 INSERTING CHARACTERS

Any pad that is in write mode automatically accepts any ASCII characters that you type at the keyboard as input to that pad. Control characters are ignored, since it is most often the case that control characters have been mapped to DM functions, although there is a way to insert the ASCII tab and End-of-File characters.

4.5.1 Inserting a Text String

The DM command **es**'*string*' inserts *string* at the current cursor position. You'll probably find this command most useful in key definition commands. For example, if you wanted to define the shifted **F1** key to insert the string *Hi there*, you would use the following key definition.

Command: `kd fls es'Hi there' ke`

This sort of key definition can be done as needed by entering the command from the DM input window. You can also put these definitions (like all key definitions) in a file, from which they can be loaded as necessary.

4.5.2 Inserting an End-of-File Mark

To insert an end-of-file mark (EOF) in a pad, type `CTRL Z` or use the DM command `eef` (insert EOF). If the line containing the cursor is empty, the DM inserts the end-of-file mark on that line. Otherwise, the DM inserts the end-of-file mark following the current line. The mark is invisible.

4.5.3 Inserting a TAB

The DOMAIN/IX key definitions file `unix_keys` redefines the shifted `TAB` key to insert an ASCII tab character. Normally, the `TAB` key simply moves the cursor to the left.

4.6 DELETING TEXT

This section deals with commands for deleting characters, words, or lines of text. To delete a larger block of text, refer to the section entitled "Cutting Text."

4.6.1 Deleting Characters

The DM command `ed`, normally mapped to `CHAR DEL`, deletes the character under the cursor. If the character under the cursor is a NEWLINE, `ed` joins the current line and the following line.

To delete the character to the left of the cursor, press `BACK SPACE`. If the pad is in overstrike mode, the `ee` command replaces the character with a blank. Both `CHAR DEL` and `BACK SPACE` are repeat keys. You can repeat the operation by holding down the key.

4.6.2 Deleting Words

The sequence of DM commands required to delete a word is normally mapped to the `F6` function key. Pressing `F6` will delete everything from the current cursor position to the next space, punctuation mark, or special character (other than a dollar sign or underscore).

`F6` invokes the following command sequence:

```
dr;/[~ a-z0-9$_]/xd
```

The DM writes the deleted word to its default paste buffer (a temporary file). You can reinsert the word elsewhere by moving the cursor to the desired location and pressing the `PASTE` key.

Note: The default paste buffer is a volatile area. It only holds the most recently deleted text object. Even a `BACK SPACE` will

wipe it out (and overwrite the buffer with whatever you backspaced over).

4.6.3 Deleting Lines

To delete text from the current cursor position to the end of the line (excluding the NEWLINE character), press the **[F7]** key. In the default key definitions file, this key is programmed to execute the following DM command sequence.

```
es ' ';ee;dr;tr;xd;tl;tr
```

The DM writes the deleted line to its default paste buffer. You can reinsert the line elsewhere by either pressing or specifying the **XP** command (see the note about the default paste buffer in the previous subsection).

4.7 DEFINING A RANGE OF TEXT

The editing commands that perform cut (delete), copy, and substitute functions operate on a range, or block, of text. You mark the beginning of a range of text by moving the cursor to the first character in that range and pressing the **[MARK]** key. Once you have marked the beginning of a range, move the cursor to the end of the range, then execute one of the DM commands that operates on a range of text. In echo mode (the default), text in the marked range will be highlighted. If you do not specify literal points, **dr** places one mark at the current cursor position.

4.8 COPYING, CUTTING, AND PASTING TEXT

The commands discussed in this section allow you to move blocks of text from one place to another in a pad, move text from one pad to another, or move text into and out of named (or default) paste buffers.

Before specifying the commands that copy or cut text, use the **dr** command or **[MARK]** to define the range of text to be copied or cut (see the previous section). If you do not define a range, the DM copies or cuts the text from the current cursor position to the end of the line.

4.8.1 Using Paste Buffers

To perform copy, cut, and paste operations, the DM uses temporary files called paste buffers. Paste buffers hold text you've copied or cut so that you can paste it in elsewhere.

You can create up to one hundred paste buffers, each containing different blocks of text. To create a paste buffer, you specify a name for the paste buffer as an argument to the commands that copy or cut text (**xc** and **xd**). To insert the contents of a paste buffer you created, specify the name of the paste buffer as an argument to the command that pastes text (**xp**). We describe the **xc**, **xd**, and **xp** commands in the next three sections.

When you log off, the DM deletes all paste buffers you created during the session. If you want to save the copied or cut text for use during another session, you can write it to a permanent file (see the **xc** and **xd** command descriptions in the next two sections).

If you do not specify the name of a paste buffer or permanent file when you specify the commands that copy or cut text, the DM writes the text to the default paste buffer. The DM also uses this default paste buffer when you press the predefined function keys and control character sequences that copy, delete, and paste text.

4.8.2 Copying Text

To copy a defined range of text from any pad into a paste buffer or file, specify the **xc** command in the following format:

Command: **xc** [*name* | **-f** *pathname*] [**-R**]

where *name* specifies the name of a paste buffer that the DM creates to hold the copied text. The **-f** option specifies the name of a permanent file for the text. For example:

Command: **xc copy_text**

copies a defined range of text into a paste buffer named *copy_text*.

Command: **xc -f copy_text**

copies a defined range of text into a permanent file named *copy_text* in the current working directory. If you supply the name of an existing paste buffer or file, **xc** overwrites its contents. If you omit the name of a paste buffer or permanent file, **xc** writes the copied text to the default (unnamed) paste buffer. The **-r** option instructs **xc** to copy a rectangular block of text that you have defined by marking a column on the left side of the text. Use the **DR** command or **MARK** column before entering the **xc -R** command. **xc** then copies all characters to the right of the specified column. By default, 880 keyboards invoke the **xc** command using the default (unnamed) paste buffer. You must specify the **xc** command with the argument or the option if you want to copy text to a named paste buffer or permanent file. Once you have copied a range of text, you can use the **xp** command to paste the text in elsewhere (see the "Pasting Text" section).

4.8.3 Cutting Text

When you cut text from a pad, the DM copies the text into a paste buffer or file and then deletes it from the pad. To cut a defined range of text, specify the command in the following format:

Command: **xd** [*name* | **-f** *pathname*] [**-R**]

where *pathname* specifies the name of a paste buffer that the DM creates to hold the deleted text. The **-f** option specifies the name of a permanent file for the text. You can use this command only in pads created with **EDIT** or via the **ce** command. If you supply the name of an

existing paste buffer or file, **xd** overwrites its contents with the newly deleted text. If you omit the name of a paste buffer or permanent file, **xd** writes the deleted text to the default (unnamed) paste buffer. The **-R** option instructs **xd** to delete a rectangular block of text, as described above under the discussion of **xc**. By default, on 880 keyboards invoke the **xd** command using the default (unnamed) paste buffer. You must specify the **xd** command with the argument or the option to write deleted text to a named paste buffer or permanent file, respectively.

Once you have cut a range of text, you can use the **xp** command (described in the next section) to paste the text in elsewhere.

4.8.4 Pasting Text

To insert the contents of a paste buffer or file into a pad at the current cursor position, specify the command in the following format:

Command: **xp** [*name* | **-f** *pathname*] [**-R**]

where *pathname* specifies the name of an existing paste buffer that contains the text you want to insert. The **-f** option specifies the name of an existing file that contains the text you want to insert. If you do not specify the name of a paste buffer or permanent file, **xp** inserts the contents of the default (unnamed) paste buffer.

The **-R** option instructs **xp** to insert a rectangular block of text that you have copied or deleted using the **xc** or **xd** command and the **-R** option. **xp** uses the current cursor position as the origin (upper left corner) of the block.

4.9 USING REGULAR EXPRESSIONS

The DM allows you to use all regular expressions supported by the AEGIS shell when doing search and substitute operations. While our regular expression structure is similar to the ones supported by the various UNIX shells and editors, it has numerous subtle differences. Read the *DOMAIN System Command Reference* and *DOMAIN System User's Guide* for full information on regular expressions. This chapter is just an overview, so we won't deal with the topic here.

4.10 SEARCHING FOR TEXT

To search from the current cursor position forward for the pattern *pat*, use the following DM command.

Command: **/pat/**

To search backward from the current cursor position for *pat*, the command syntax is

Command: **\pat**

In either case, the *pat* may be a regular expression.

A search operation moves the cursor to the first character in the specified *pat*. If necessary, the pad moves under the window to display the matching string. If the search fails, the cursor position does not change, and the DM displays the message

No match

in its output window.

Searches do not wrap around the end or beginning of the file. Therefore, to search an entire pad, you should position the cursor at the beginning of the pad.

4.10.1 Case Sensitivity

By default, searches are not case sensitive. To perform a case-sensitive search, you must first set case sensitivity on by executing the DM command

Command: **sc -on**

If the DM scans more than 100 lines in a search operation, it displays a Searching...

message in its output window, then polls for keystrokes after every 10 lines searched. You may cancel the search by typing **CTRL X** or by pressing a key that has been defined to invoke the **abrt** or **sq** command.

To repeat the last forward search, use the command

Command: **//**

To repeat the last backward search, use the command

Command: ****

The DM saves the most recent search instruction, so you may repeat it even if you have specified other (non-searching) commands since then.

4.10.2 Cancelling a Search Operation

To cancel the current search operation, type **↑X**, mapped to the **abrt** command. Since you cannot type DM commands for the pad being searched, you must use **CTRL X** or define a key to invoke **abrt**. (See the "Defining Keys" section in the *DOMAIN System User's Guide*.)

The DM command also cancels a search operation. As with the **abrt** command, you must define a key to invoke **sq** during a search. When you type **↑X** or press a key defined to invoke **abrt** or **sq**, the DM displays the message "Search aborted" in its output window.

4.11 SUBSTITUTING TEXT

Unlike searches, which ignore case unless told otherwise, all substitutions are case-sensitive. If the DM scans more than 100 lines while processing a substitute command, it displays a

Substitute in progress...

message in its output window. Then it polls for keystrokes after every 10 lines it processes.

4.11.1 Substituting All Occurrences of a String

To replace all occurrences of a *pat* with a *replacement*, use the following DM syntax:

Command: `s[[/[pat]]/replacement/]`

Regular expressions are allowed in the *pat*, but not in the *replacement*. An ampersand (&) in *replacement* expands to *pat*. For example

Command: `s/Tom/& Smith/`

replaces all occurrences of "Tom" with "Tom Smith" over the defined range of text. The `s` command does not move the cursor or the pad, even if it makes changes in areas of the pad not visible through the window.

4.11.2 Substituting the First Occurrence of a String

The `so` command is like `s` except that `so` replaces only the first occurrence of a string in each line of a defined range of text.

4.11.3 Changing the Case of Letters

To change the case of letters in a defined range of text, specify the command in the following format:

Command: `case [-s | -u | -l]`

where `-s` swaps all uppercase letters for lowercase and all lowercase letters for uppercase, `-u` forces all letters to uppercase, and `-l` forces all letters to lowercase.

4.12 UNDOING PREVIOUS COMMANDS

To undo the most recent DM command you entered, use the `UNDO` key (mapped to the DM command `undo`).

The `undo` command works by compiling a history of DM operations in input and edit pads in reverse chronological order. `UNDO` reverses the effect of the most recent DM command you specified. Successive `UNDO`s reverse DM commands further back in history.

To compile its history of activities, the DM uses undo buffers (one per edit pad and one per input pad). The undo buffers are circular lists that, when full, eliminate the oldest entries to make room for new ones, so

that in practice, you may not be able to **undo** everything. The DM groups entries together in sets. For example, a S (SUBSTITUTE) command may change five lines. While the DM considers this to be five entries, the five entries are grouped into a single set so that one UNDO will change all five lines back to their original state. When a buffer becomes full, the DM erases the oldest of entries. This means that UNDO will never partially undo an operation; it will either completely undo it or do nothing. An undo buffer for an edit pad can hold up to 1024 entries. An undo buffer for an input pad can hold up to 128 entries.



Index

?name, ed error message	1-5	<	2-17
., used by ed	1-7	>	2-17
A		abbreviate	2-7
append, ed command	1-2	append	2-7
args, ex command	2-3	args	2-7
autoindent, vi option	3-29	change	2-7
B		copy	2-7
buffer, used by ed	1-2	delete	2-8
buffers, named, in vi	3-16	edit	2-8
C		exit	2-16
case sensitivity		file	2-8
in DM editor	4-9	global	2-9
in vi searches	3-29	insert	2-10
context search, in vi	3-7	join	2-10
control characters		list	2-10
displayed by vi	3-14	map	2-10
to type in vi	3-30	mark	2-10
counts, in vi commands	3-9, 3-15, 3-26	move	2-10
cut-and-paste		n	2-11
using DM editor	4-7	next	2-11
using ed	1-15	number	2-11
DM edit pad		open	2-11
to close	4-3	preserve	2-11
to save	4-3	print	2-11
DM editor		put	2-11
.bak files	4-3	quit	2-12
insert mode	4-4	read	2-12
dot, used by ed	1-7	recover	2-12
E		rewind	2-13
environment variables		set	2-13
EXINIT	2-1, 3-20	shell	2-13
TERM	3-4	source	2-13
ESC, in vi	3-5	stop	2-14
ex		substitute	2-13
command abbreviations	2-4	ta	2-14
command mode	2-4	unabbreviate	2-14
counts	2-4	undo	2-14
report option	2-5	unmap	2-15
Shell escape	2-17	version	2-15
text input mode	2-4	visual	2-15
to scroll	2-18	write	2-15
ex commands		yank	2-16
		z	2-16
		join	2-10
		EXINIT, environment variable	2-1, 3-20
		exrc, options allowed in	2-20

I			
insert mode, in vi	3-10		
M			
magic/nomagic, in vi	3-29		
metacharacters, used by ed	1-17		
N			
next, ex command	2-3		
P			
paste buffer			
DM default	4-6		
DM named	4-6		
regular expressions			
in ex	2-18		
in vi	3-29		
S			
scroll, in ex	2-18		
Shell escape			
from ex	2-17		
from vi	3-17		
shiftwidth, ex option	2-17		
substitute, ed command	1-9		
T			
tab character, in vi	3-14		
TERM environment variable, in vi	3-4		
TERMCAP, used by ex/vi	2-1		
tset, to specify terminal type	3-4		
U			
undo			
in DM editor	4-11		
in ex	2-14		
in vi	3-13		
V			
vi, alternate file name	3-27		
vi commands			
% (to matching delimiter)	3-35		
((to next sentence)	3-35		
) (to prev. sentence)	3-35		
+ (head of next line)	3-33		
- (head of prev. line)	3-33		
0 (to left margin)	3-33		
? (search backward)	3-37		
[[(to beginning of this section)	3-35		
' (goto marked line)	3-36		
\ (goto mark)	3-36		
`` (to previous context mark)	3-36		
↑B (to previous page)	3-35		
↑D (scroll down)	3-34		
↑F (to next page)	3-34		
↑U (scroll up)	3-34		
]] (to next section)	3-35		
" (prev. context)	3-8		
" (to previous context line)	3-36		
a (append)	3-37		
b (back one word)	3-34		
e (end of word)	3-34		
G (go to)	3-34		
H (home)	3-35		
h (move left)	3-33		
J (join)	3-22		
j (move down)	3-33		
k (move up)	3-33		
kill (erase line)	3-30		
l (move right)	3-33		
L (to last screen line)	3-36		
m (mark)	3-36		
M (to middle of screen)	3-36		
n (next)	3-28		
p (put)	3-16		
p (put)	3-39		
to write/quit	3-6		
u (undo)	3-41		
w (to next word)	3-34		
y (yank)	3-16		
y (yank)	3-39		
ZZ	3-28		
ZZ	3-6		
{ (to beginning of paragraph)	3-35		
} (to next paragraph.)	3-35		
G (Go to)	3-8		
vi options			
autoindent	3-22		
autowrite	3-28		

CONTENTS

1. A troff Tutorial 1-1
 - 1.1 PREPARING AN INPUT FILE 1-2
 - 1.2 POINT SIZES AND LINE SPACING 1-2
 - 1.3 FONTS AND SPECIAL CHARACTERS 1-5
 - 1.4 INDENTATION AND LINE LENGTH 1-6
 - 1.5 TABS 1-8
 - 1.6 LOCAL MOTIONS, LINES, AND CHARACTERS 1-9
 - 1.7 STRINGS 1-12
 - 1.8 INTRODUCTION TO MACROS 1-13
 - 1.9 TITLES, PAGES, AND NUMBERING 1-14
 - 1.9.1 Page Numbers 1-16
 - 1.10 NUMBER REGISTERS AND ARITHMETIC 1-16
 - 1.11 MACROS WITH ARGUMENTS 1-18
 - 1.12 CONDITIONALS 1-20
 - 1.13 ENVIRONMENTS 1-22
 - 1.14 DIVERSIONS 1-22
2. The troff Reference Manual 2-1
 - 2.1 INTRODUCTION 2-1
 - 2.1.1 Usage 2-1
 - 2.1.2 Input File Format 2-3
 - 2.1.3 Output Device Resolution 2-4
 - 2.1.4 Numerical Parameter Input 2-4
 - 2.1.5 Numerical Expressions 2-5
 - 2.1.6 Notational Conventions 2-5
 - 2.2 FONT AND CHARACTER SIZE CONTROL 2-5
 - 2.2.1 Character Set 2-5
 - 2.2.2 Fonts 2-6
 - 2.2.3 Character Size 2-6
 - 2.3 PAGE CONTROL 2-8
 - 2.4 TEXT FILLING AND ADJUSTING 2-10
 - 2.4.1 Filling and Adjusting 2-10
 - 2.4.2 Interrupted Text 2-11
 - 2.5 VERTICAL SPACING 2-12
 - 2.5.1 Baseline Spacing 2-12
 - 2.5.2 Extra Line Space 2-12
 - 2.5.3 Blocks of Vertical Space 2-13
 - 2.6 LINE LENGTH AND INDENTING 2-14
 - 2.7 MACROS, STRINGS, DIVERSIONS, TRAPS 2-14
 - 2.7.1 Macros and Strings 2-14
 - 2.7.2 Copy Mode Input Interpretation 2-15
 - 2.7.3 Arguments 2-15
 - 2.7.4 Diversions 2-16
 - 2.7.5 Traps 2-17
 - 2.8 NUMBER REGISTERS 2-19
 - 2.9 TABS, LEADERS, AND FIELDS 2-20

- 2.9.1 Tabs and Leaders 2-20
- 2.9.2 Fields 2-21
- 2.10 CONVENTIONS AND TRANSLATIONS 2-21
 - 2.10.1 Input Character Translations 2-21
 - 2.10.2 Ligatures 2-22
 - 2.10.3 Backspacing, Underlining, Overstriking 2-22
 - 2.10.4 Request Characters 2-23
 - 2.10.5 Output Translation 2-23
 - 2.10.6 Transparent Throughput 2-24
 - 2.10.7 Comments and Concealed Newlines 2-24
- 2.11 LOCAL MOTIONS 2-24
 - 2.11.1 Local Motions 2-24
 - 2.11.2 The Width Function 2-25
 - 2.11.3 The Horizontal Place Marker 2-25
- 2.12 OVERSTRIKES, BRACKETS, AND LINES 2-25
 - 2.12.1 Overstriking 2-25
 - 2.12.2 Zero-Width Characters 2-25
 - 2.12.3 Large Brackets 2-25
 - 2.12.4 Line Drawing 2-26
- 2.13 HYPHENATION 2-27
- 2.14 THREE-PART TITLES 2-28
- 2.15 OUTPUT LINE NUMBERING 2-28
- 2.16 CONDITIONAL ACCEPTANCE OF INPUT 2-29
 - 2.16.1 Built-In Conditions 2-30
- 2.17 ENVIRONMENTS 2-30
- 2.18 INSERTIONS FROM STANDARD INPUT 2-31
 - 2.18.1 Prompts 2-31
- 2.19 INPUT/OUTPUT FILE SWITCHING 2-32
- 2.20 MISCELLANEOUS REQUESTS 2-32
- 2.21 OUTPUT AND ERROR MESSAGES 2-33
- 2.22 FONT STYLE EXAMPLES 2-34
- 2.23 INPUT CHARACTER NAMES 2-35
 - 2.23.1 Special Characters on Standard Fonts 2-35
 - 2.23.2 Characters on the Special Font 2-35
- 2.24 SUMMARY OF REQUESTS 2-38
 - 2.24.1 Font and Character Size Control 2-38
 - 2.24.2 Page Control 2-39
 - 2.24.3 Text Filling, Adjusting, and Centering 2-39
 - 2.24.4 Vertical Spacing 2-39
 - 2.24.5 Line Length and Indenting 2-40
 - 2.24.6 Macros, Strings, Diversions, Traps 2-40
 - 2.24.7 Number Registers 2-40
 - 2.24.8 Tabs, Leaders, and Fields 2-41
 - 2.24.9 I/O Conventions and Translations 2-41
 - 2.24.10 Hyphenation 2-41
 - 2.24.11 Three Part Titles 2-42
 - 2.24.12 Output Line Numbering 2-42
 - 2.24.13 Conditional Acceptance of Input 2-42

- 2.24.14 Environment Switching 2-43
- 2.24.15 Insertions from the Standard Input 2-43
- 2.24.16 Input/Output File Switching 2-43
- 2.24.17 Miscellaneous Requests 2-43
- 2.25 SUMMARY OF ESCAPE SEQUENCES 2-44
- 2.26 SUMMARY OF PRE-DEFINED GENERAL NUMBER
REGISTERS 2-45
- 2.27 SUMMARY OF PRE-DEFINED READ-ONLY
NUMBER REGISTERS 2-45
- 3. The -ms Macro Package 3-1
 - 3.1 INTRODUCTION 3-1
 - 3.2 COVER SHEETS AND FIRST PAGES 3-1
 - 3.3 PARAGRAPHS 3-2
 - 3.4 PAGE HEADINGS 3-2
 - 3.5 MULTI-COLUMN FORMATS 3-2
 - 3.6 HEADINGS 3-3
 - 3.7 INDENTED PARAGRAPHS 3-4
 - 3.8 EMPHASIS 3-6
 - 3.9 FOOTNOTES 3-7
 - 3.10 DISPLAYS AND TABLES 3-8
 - 3.11 BOXING WORDS OR LINES 3-8
 - 3.12 KEEPING BLOCKS TOGETHER 3-9
 - 3.13 NROFF/TROFF REQUESTS 3-9
 - 3.14 DATE 3-9
 - 3.15 REGISTERS 3-9
 - 3.16 ACCENTS 3-10
 - 3.17 PRINTING 3-11
 - 3.18 TYPESETTING MATHEMATICS 3-12
 - 3.19 BIBLIOGRAPHY ENTRIES 3-12
 - 3.20 TABLE OF CONTENTS 3-13
 - 3.21 SUMMARY OF -ms MACROS 3-14
 - 3.22 SUMMARY OF -ms REGISTER NAMES 3-14
- 4. The -me Macro Package 4-1
 - 4.1 INTRODUCTION 4-1
 - 4.2 PARAGRAPHING 4-2
 - 4.3 SECTION HEADINGS 4-2
 - 4.4 HEADERS AND FOOTERS 4-4
 - 4.5 DISPLAYS 4-5
 - 4.6 ANNOTATIONS 4-7
 - 4.7 MULTI-COLUMN OUTPUT 4-8
 - 4.8 FONTS AND SIZES 4-8
 - 4.9 ROFF SUPPORT 4-9
 - 4.10 PREPROCESSOR SUPPORT 4-10
 - 4.11 MISCELLANEOUS MACROS 4-10
 - 4.12 STANDARD PAPERS 4-11
 - 4.13 PRE-DEFINED STRINGS 4-13
 - 4.14 SPECIAL CHARACTERS AND MARKS 4-13

- 5. The -mm Macro Package 5-1
 - 5.1 INTRODUCTION 5-1
 - 5.2 CONVENTIONS 5-1
 - 5.3 INPUT FILE STRUCTURE 5-1
 - 5.4 FORMATTERS 5-2
 - 5.5 INVOKING THE MACROS 5-3
 - 5.5.1 The mm Command 5-3
 - 5.5.2 Using the -cm or -mm Flag 5-4
 - 5.5.3 Typical Command Lines 5-4
 - 5.5.4 Parameters Set on the Command Line 5-6
 - 5.5.5 Omission of -cm or -mm 5-8
 - 5.6 FORMATTING CONCEPTS 5-9
 - 5.6.1 Basic Terms 5-9
 - 5.6.2 Arguments and Double Quotes 5-9
 - 5.6.3 Unpaddable Spaces 5-10
 - 5.6.4 Hyphenation 5-10
 - 5.6.5 Tabs 5-11
 - 5.6.6 Special Use of the BEL Character 5-11
 - 5.6.7 Bullets 5-12
 - 5.6.8 Dashes, Minus Signs, and Hyphens 5-12
 - 5.6.9 Trademark String 5-12
 - 5.7 PARAGRAPHS AND HEADINGS 5-13
 - 5.7.1 Paragraphs 5-13
 - 5.7.2 Numbered Headings 5-14
 - 5.7.3 Altering Heading Pre-Space 5-15
 - 5.7.4 Heading Post-Space 5-15
 - 5.7.5 Centered Headings 5-16
 - 5.7.6 Format Control by Heading Level 5-16
 - 5.7.7 Nroff Heading Underlining Styles 5-16
 - 5.7.8 Heading Point Sizes 5-16
 - 5.7.9 Marking Styles 5-17
 - 5.7.10 Unnumbered Headings 5-18
 - 5.7.11 Headings and the Table of Contents 5-18
 - 5.7.12 Page Numbering Style 5-18
 - 5.7.13 User Exit Macros 5-19
 - 5.7.14 Hints for Large Documents 5-21
 - 5.8 LISTS 5-21
 - 5.8.1 The Parts of a List 5-21
 - 5.8.2 Sample Nested Lists 5-22
 - 5.8.3 Common List Macros 5-23
 - 5.8.3.1 List Item 5-23
 - 5.8.3.2 List End 5-24
 - 5.8.4 List Initialization Macros 5-24
 - 5.8.4.1 Numbered and Alphabetized Lists 5-24
 - 5.8.4.2 Bulleted List 5-25
 - 5.8.4.3 Dashed List 5-25
 - 5.8.4.4 Marked List 5-25
 - 5.8.4.5 Reference List 5-26

- 5.8.4.6 Variable-Item List 5-26
- 5.8.5 List-Begin Macro and Customized Lists 5-27
- 5.8.6 User-Defined List Structures 5-29
- 5.9 DISPLAYS 5-31
 - 5.9.1 Static Displays 5-32
 - 5.9.2 Floating Displays 5-33
 - 5.9.3 Tables 5-35
 - 5.9.4 Equations 5-37
 - 5.9.5 Captions 5-37
 - 5.9.6 List of Figures, Tables, Etc. 5-38
- 5.10 FOOTNOTES 5-38
 - 5.10.1 Automatic Numbering of Footnotes 5-38
 - 5.10.2 Delimiting Footnote Text 5-38
 - 5.10.3 Format of Footnote Text 5-39
 - 5.10.4 Spacing Between Footnote Entries 5-40
- 5.11 PAGE HEADERS AND FOOTERS 5-40
 - 5.11.1 Default Headers and Footers 5-41
 - 5.11.2 Page Header 5-41
 - 5.11.3 Even-Page Header 5-41
 - 5.11.4 Odd-Page Header 5-42
 - 5.11.5 Page Footer 5-42
 - 5.11.6 Even-Page Footer 5-42
 - 5.11.7 Odd-Page Footer 5-42
 - 5.11.8 Footer on the First Page 5-42
 - 5.11.9 Section-Page Numbering 5-42
 - 5.11.10 Strings and Registers in Headers/Footers 5-42
 - 5.11.11 Header and Footer Example 5-43
 - 5.11.12 Generalized Top-of-Page Processing 5-43
 - 5.11.13 Generalized Bottom-of-Page Processing 5-44
 - 5.11.14 Top and Bottom Margins 5-44
 - 5.11.15 Private Documents 5-45
- 5.12 TABLE OF CONTENTS AND COVER SHEET 5-45
 - 5.12.1 Table of Contents 5-45
- 5.13 REFERENCES 5-47
 - 5.13.1 Automatic Numbering of References 5-47
 - 5.13.2 Delimiting Reference Text 5-48
 - 5.13.3 Subsequent References 5-48
 - 5.13.4 Reference Page 5-48
- 5.14 MISCELLANEOUS FEATURES 5-49
 - 5.14.1 Bold, Italic, and Roman Type 5-49
 - 5.14.2 Justification of Right Margin 5-50
 - 5.14.3 SCCS Release Identification 5-50
 - 5.14.4 Two-Column Output 5-50
 - 5.14.5 Column Headings 5-51
 - 5.14.6 Vertical Spacing 5-52
 - 5.14.7 Skipping Pages 5-53
 - 5.14.8 Forcing an Odd Page 5-53
 - 5.14.9 Setting Point Size and Vertical Spacing 5-53

- 5.14.10 Producing Accents 5-54
- 5.14.11 Inserting Text Interactively 5-54
- 5.14.12 Bell Labs Macros 5-55
- 5.14.13 Date and Format Changes 5-55
- 5.14.14 "Copy to" and Other Notations 5-56
- 5.14.15 Approval Signature Line 5-57
- 5.14.16 Forcing a One-Page Letter 5-57
- 5.15 ERRORS AND DEBUGGING 5-57
 - 5.15.1 Error Terminations 5-57
 - 5.15.2 Disappearance of Output 5-58
 - 5.15.3 MM Error Messages 5-58
 - 5.15.4 Formatter Error Messages 5-60
- 5.16 EXTENDING AND MODIFYING THE MACROS 5-61
 - 5.16.1 Naming Conventions 5-61
 - 5.16.2 Names Used by Formatters 5-61
 - 5.16.3 Names Used by -mm 5-61
 - 5.16.4 Names Used by eqn, neqn, and tbl 5-62
 - 5.16.5 User-Definable Names 5-62
- 5.17 SAMPLE EXTENSIONS 5-62
 - 5.17.1 Appendix Headings 5-62
 - 5.17.2 Hanging Indent with Tabs 5-63
- 5.18 SUMMARY OF MACROS, STRINGS, AND NUMBER REGISTERS 5-64
 - 5.18.1 Macros 5-64
 - 5.18.2 Strings 5-69
 - 5.18.3 Number Registers 5-70
- 6. Eqn — a Pre-Processor for Text With Equations 6-1
 - 6.1 INTRODUCTION 6-1
 - 6.2 DISPLAYED EQUATIONS 6-1
 - 6.3 SPACES AND NEWLINES 6-2
 - 6.3.1 Input Spaces 6-2
 - 6.3.2 Output Spaces 6-3
 - 6.4 SYMBOLS, SPECIAL NAMES, GREEK 6-3
 - 6.5 DELIMITING SPECIAL SEQUENCES 6-3
 - 6.6 SUBSCRIPTS AND SUPERSSCRIPTS 6-4
 - 6.7 BRACES FOR GROUPING 6-4
 - 6.8 FRACTIONS 6-5
 - 6.9 SQUARE ROOTS 6-6
 - 6.10 SUMMATION, INTEGRAL, ETC. 6-6
 - 6.11 SIZE AND FONT CHANGES 6-7
 - 6.12 DIACRITICAL MARKS 6-8
 - 6.13 QUOTED TEXT 6-9
 - 6.14 LINING UP EQUATIONS 6-9
 - 6.15 LARGE DELIMITERS 6-10
 - 6.16 PILES 6-11
 - 6.17 MATRICES 6-12
 - 6.18 SHORTHAND FOR IN-LINE EQUATIONS 6-12
 - 6.19 DEFINITIONS 6-13

6.20	LOCAL MOTIONS	6-14
6.21	A LARGE EXAMPLE	6-14
6.22	KEYWORDS, PRECEDENCES, ETC.	6-15
6.23	TROUBLESHOOTING	6-17
7.	Tbl — a Preprocessor for Formatting Tables	7-1
7.1	INTRODUCTION	7-1
7.2	INPUT	7-1
7.3	GLOBAL OPTIONS	7-2
7.4	FORMAT KEY-LETTERS	7-3
7.4.1	Horizontal Lines	7-5
7.4.2	Vertical Lines	7-5
7.4.3	Space Between Columns	7-5
7.4.4	Vertical Spanning	7-5
7.4.5	Font Changes	7-5
7.4.6	Point Size Changes	7-5
7.4.7	Vertical Spacing Changes	7-6
7.4.8	Column Width Indication	7-6
7.4.9	Equal Width Columns	7-6
7.4.10	Alternative Notation	7-6
7.4.11	Defaults	7-6
7.5	DATA	7-7
7.5.1	Troff Requests Within Tables	7-7
7.5.2	Full Width Horizontal Lines	7-7
7.5.3	Single Column Horizontal Lines	7-7
7.5.4	Short Horizontal Lines	7-7
7.5.5	Vertically Spanned Items	7-7
7.5.6	Text Blocks	7-7
7.6	ADDITIONAL COMMAND LINES	7-8
7.7	USAGE	7-9
7.8	EXAMPLES	7-11
7.9	SUMMARY OF COMMANDS AND KEY-LETTERS	7-21



Chapter 1: A troff Tutorial

Troff is a text-formatting program that produces output suitable for various typesetting devices, including laser printers and phototypesetters. This chapter (and, in fact, this entire book) is an example of **troff** output.

Most laser printers and phototypesetters make use of at least four fonts: Roman, Italic, bold, and Greek, as well as a number of special characters and mathematical symbols. Characters can be printed in a range of sizes, and placed anywhere on the page.

Troff gives you full control over fonts, sizes, and character positions, as well as the usual features of a formatter: right-margin justification, automatic hyphenation, page titling and numbering, and so on. It also provides macros, arithmetic variables and operations, and conditional testing for complicated formatting tasks.

This chapter is an introduction to the fundamentals of **troff**. It presents just enough information to help you with simple formatting tasks, and to get you started with existing macro packages. (See Chapters 3 and 4). The other popular UNIX text formatter, **nroff**, has much in common with **troff**. This chapter serves as a tutorial on both.

In many ways, **troff** resembles an assembly language, a remarkably powerful and flexible one, but one in which many operations must be specified in precise detail, and in a form that is too hard for most people to use effectively. In most cases, it is far easier to access **troff**'s facilities through some intermediary, such as a pre-processor or a macro package.

There are two widely-used pre-processors, each of which has its own set of commands. **eqn** provides an easy-to-learn language for typesetting mathematics; the **eqn** user does not need to know any **troff** commands to typeset complex mathematical equations. The **tbl** pre-processor provides the same convenience for producing tables of arbitrary complexity. Chapters 6 and 7 of this section deal with **eqn** and **tbl**, respectively.

For producing normal text (which may well contain mathematics or tables), there are a number of macro packages that define formatting rules and operations for specific styles of documents, and reduce the amount of direct contact with **troff** to an acceptable level. In particular, the **-ms** and **-mm** packages provide most of the facilities needed for a wide range of document preparation tasks. (This document was prepared using **-mm**.) You will probably find these packages easier to use than "bare" **troff** once you get beyond the most trivial operations; you should always consider them first. Chapters 3, 4, and 5 of this section deal with macro packages.

In the few cases where existing macro packages don't do the whole job, it's not too difficult to customize the macro packages that already exist. This is a far easier chore than writing a macro package from scratch.

In this chapter, we will describe only the more useful parts of **troff**. Our emphasis is on showing how to do simple things, and how to make incremental changes to what already exists.

1.1 PREPARING AN INPUT FILE

To use **troff** you have to embellish the actual text you want printed with additional information that tells **troff** *how* you want it printed. This information comes in two principal forms: requests, which are built in to the **troff** program itself, and macros, which are built up from multiple requests. The typical **troff** request is placed on a line by itself, separated from the surrounding text by newlines, and begins with a period. For example,

```
Some text.
.ps 16
Some more text.
```

will change the "point size", that is, the size of the letters being printed, to "16 point" (one point is 1/72 inch) like this:

```
Some text. Some more text.
```

Occasionally, though, size changes and other things have to occur in the middle of a line. For example, to produce

$$\text{Area} = \pi r^2$$

you have to type

```
Area = \(*p\flr\fr\|\s8\u2\d\s0
```

(the meaning of which we will explain shortly). The backslash character `\` is used to introduce **troff** commands and special characters within a line of text.

1.2 POINT SIZES AND LINE SPACING

As mentioned above, the command sets the point size. One point is 1/72 inch, so 6-point characters are at most 1/12 inch high, and 36-point characters are about six times as large (1/2 inch). There are 15 point sizes available, as shown below.

6 point: Pack my box with five dozen liquor jugs.
 7 point: Pack my box with five dozen liquor jugs.
 8 point: Pack my box with five dozen liquor jugs.
 9 point: Pack my box with five dozen liquor jugs.
 10 point: Pack my box with five dozen liquor
 11 point: Pack my box with five dozen
 12 point: Pack my box with five dozen
 14 point: Pack my box with five
 16 point 18 point 20 point

22 24 28 36

If the number after `.ps` is not one of these sizes, **troff** rounds it up to the next value, with a maximum of 36. If no number follows `.ps`, **troff** reverts to the previous size. **Troff** is initialized to a point size of 10. This document is set in 12 point type.

The point size can also be changed in the middle of a line or even a word with the in-line command `\s`. To produce

DOMAIN/IX is the DOMAIN system's distributed UNIX environment.

you could put the following line in your input file.

DOMAIN/IX is the \s10DOMAIN\s0
 system's distributed \s10UNIX\s0 environment.

The `\s` request should be followed by a legal point size or a zero (which causes the point size to revert to its previous value).

Relative size changes are also legal and useful:

`\s-2SMALL CAPS\s+2`

temporarily decreases the current size by two points to obtain SMALL CAPS, then restores it. Relative size changes have the advantage that the size difference is independent of the starting size of the document. The amount of the relative change is restricted to a single digit.

Note: You cannot request a relative size change greater than 9 points.

Another thing that determines the look of the type is the leading, or spacing between lines. **Troff** allows you to specify vertical spacing independently of point size. Vertical spacing is measured from the bottom of one line to the bottom of the next. The command to control vertical spacing is `.vs`. For running text, it is usually best to set the vertical spacing about 20% bigger than the character size. This book was set in "12 on 13.5," that is,

.ps 12
.vs 13.5p

If we changed to

.ps 9
.vs 9p

the running text would look like this. After reading a few lines, you will agree it looks a little cramped. Vertical spacing is partly a matter of taste, depending on how much text you want to squeeze into a given space, and partly a matter of traditional printing style. By default, **troff** uses 10 on 12.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. This is 12 on 13.5.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. For example, 10 on 12 uses about twice as much space as 7 on 8. This is 6 on 7, which is even smaller. It packs a lot more words per line, but you can go blind trying to read it.

When used without arguments, **.ps** and **.vs** revert to the previous (or default) size and vertical spacing respectively.

The command **.sp** is used to get extra vertical space. Unadorned, it gives you one extra blank line (one **.vs**, whatever that has been set to). Typically, that's more or less than you want, so **.sp** can be followed by information about how much space you want:

.sp 2i

means "two inches of vertical space,"

.sp 2p

means "two points of vertical space"; and

.sp 2

means "two vertical spaces," two of whatever **.vs** is set to (this can also be made explicit with **.sp 2v**); **troff** also understands decimal fractions in most places, so

.sp 1.5i

is a space of 1.5 inches. These same scale factors can be used after **.vs** to define line spacing, and after most commands that deal with physical dimensions.

It should be noted that **troff** converts all size numbers to units of 1/432 inch (1/6 point), the resolution of the original typesetting machinery for which **troff** was written. For most purposes, this is enough resolution that you don't have to worry about accuracy of representation. The situation is not quite so good vertically, where resolution is 1/144 inch (1/2 point).

1.3 FONTS AND SPECIAL CHARACTERS

Troff and the typesetter allow four different fonts at any one time. Normally, three fonts (Roman, Italic and bold) and one collection of special characters are permanently mounted. **Troff** prints in the Roman font unless told otherwise. To switch into bold, use the **.ft** command

```
.ft B
```

and for italics, use

```
.ft I
```

To return to roman, use

```
.ft R
```

and to return to the previous font, whatever it was, use either

```
.ft P
```

or just **.ft** with no argument. The “underline” request

```
.ul
```

causes the next input line to print in italics. A **.ul** request can be followed by a count to indicate that more than one line is to be italicized.

Fonts can also be changed within a line or even within a word by using the in-line command **\f**

```
boldface text
```

is produced by

```
\fBbold\fIface\fR text
```

If you want to do this so the previous font is left undisturbed, insert extra **\fP** commands, like this:

```
\fBbold\fP\fIface\fP\fR text\fP
```

Because only the immediately previous font is remembered, you have to restore the previous font after each change or you can lose it. The same is true of **.ps** and **.vs** when used without an argument.

There are other fonts available besides the standard set, although you can still use only four at any given time. The command **.fp** tells **troff** what fonts are physically mounted on the typesetter:

```
.fp 3 H
```

says that the Helvetica font is mounted on position 3. If you do not plan to use the standard fonts, appropriate **.fp** commands should appear at the beginning of your document, before the first text line.

It is possible to make a document relatively independent of the actual fonts used to print it by using font numbers instead of names; for example, **\f3** and **.ft 3** mean “whatever font is mounted at position 3”, and

thus work for any setting. Normal settings are roman font on 1, italic on 2, bold on 3, and special on 4.

Special characters have four-character names beginning with `\(`, and they may be inserted anywhere. For example,

$\frac{1}{4}$

is produced by

`\(14`

In particular, Greek letters are all of the form `\(*-`, where `-` is the English name of the Greek letter you wish to print. Upper-case Greek letters are specified by capitalizing the first letter of the English name. Thus to get

$\Sigma(\alpha \times \beta) \rightarrow \infty$

in bare **troff**, we have to type

`\(*S\(*a\(\mu\(*b) \(-> \(\if`

There is a complete list of these special character names at the end of this chapter.

In **eqn**, the same effect can be achieved with the input

`SIGMA (alpha times beta) -> inf`

which is less concise, but clearer to the uninitiated.

Notice that each four-character name is a single character as far as **troff** is concerned; the “translate” command

`.tr \(\mi \(\em`

is perfectly clear, meaning

`.tr - —`

that is, to translate a minus sign (`-`) in the input file to a one em dash (`—`) upon output.

Some characters are automatically translated into others: grave and acute accents become opening and closing single quotes. The combination of “...” is generally preferable to the double quotes ”...”. Similarly, a typed minus sign becomes a hyphen `-`. To print an explicit minus sign, use `\-`. To get a backslash printed, use `\e`.

1.4 INDENTATION AND LINE LENGTH

Troff is initialized to a line length of 6.5 inches. To reset the line length, use the `.ll` command, as in

`.ll 6i`

As with `.sp`, the actual length can be specified in several ways; inches are

probably the most familiar, though people who set type often prefer to specify in *picas*.

The maximum line length allowed by **troff** is 7.5 inches. To use the full width, you will have to reset the default physical left margin (“page offset”), which is normally slightly less than one inch from the left edge of the paper. This is done with the **.po** command.

```
.po 0
```

sets the offset as far to the left as it will go.

The indent command **.in** indents the left margin a specified amount from the page offset. If we use **.in** to expand (indent) the left margin and **.ll** to “expand” the right margin (by reducing the line length), we can set off blocks of text:

```
.in 1i
.ll -1.0i
Pater noster . . . .
.ll +1.0i
.in -1.0i
```

will create a block that looks like this:

```
Pater noster qui est in caelis sanctificetur
nomen tuum; adveniat regnum tuum; fiat
voluntas tua, sicut in caelo, et in terra. . . .
Amen
```

Notice the use of “+” and “-” to specify the amount of change. These change the previous setting by the specified amount, rather than just overriding it. The distinction is quite important: **.ll +1i** makes lines one inch longer; **.ll 1i** makes them one inch long.

With **.in**, **.ll** and **.po**, the previous value is used if no argument is specified.

To indent a single line, use the “temporary indent” command **.ti**. For example, one way to make an indented paragraph would be to precede the paragraph text with the following request.

```
.ti 3
```

Three of what? The default unit for **.ti**, as for most horizontally oriented commands **.ll**, (**.in**, **.po**), is ems. An em is roughly the width of the letter “m” in the current point size. (Strictly speaking, an em in size *p* is *p* points wide.) Although inches are usually clearer than ems to people who don’t set type for a living, ems have a place; they are a measure of size that is proportional to the current point size. If you want to make text that keeps its proportions regardless of point size, you should use ems for all dimensions. Ems can be specified as scale factors directly, as in **.ti 2.5m**.

Lines can also be indented negatively if the indent is already positive:

```
.ti -0.7i
```

causes the next line to be moved back seven-tenths of an inch. To make a decorative initial capital, we indent the whole paragraph, then move the letter “P” back with a `.ti` command:

```
Pater noster qui est in caelis sanctificetur nomen
tuum; adveniat regnum tuum; fiat voluntas tua, sicut
in caelo, et in terra. ... Amen.
```

Of course, there is also some work needed to make the “P” bigger (just a “`\s24P\s0`”), and to move it down from its normal position (see the section on local motions).

1.5 TABS

Tabs (the ASCII “horizontal tab” character) can be used to produce output in columns, or to set the horizontal position of output. Most people use tabs only in unfilled text. **Troff** tab stops are set by default every half inch from the current indent. Tab settings can be changed by the `.ta` command. The **troff** request line below sets a tab stop every inch.

```
.ta 1i 2i 3i 4i 5i 6i
```

These tab stops are left-justified, like tab stops on a typewriter. This can make for tedious going if you have to line up columns of right-justified numbers in a table. The **troff** pre-processor `tbl`, described in Chapter 7 of this section, allows much more detailed specification of tabular layouts. Use it whenever you have large or complex tables to format.

For a handful of numeric columns, you can do it this way. Precede every number by enough blanks to make the number line up when typed.

```
.nf
.ta 1i 2i 3i
  1 tab  2 tab  3
 40 tab 50 tab 60
700 tab 800 tab 900
.fi
```

Then, change each leading blank into the string `\0`. This is an “escape” character that does not print, but that has the same width as a digit. When printed, the input in the example above will produce

```
  1  2  3
 40 50 60
700 800 900
```

It is also possible to fill up tabbed-over space with some character other than blanks. To specify a different “tab replacement character,” use the `.tc` command (`\(ru is ”_”`):

```
.ta 1.5i 2.5i
.tc \ (ru
Name tab Age tab
```

produces

```
Name _____ Age _____
```

To reset the tab replacement character to a blank, use `.tc` with no argument. (Lines can also be drawn with the `\l` command, described in a later part of this chapter.)

1.6 LOCAL MOTIONS, LINES, AND CHARACTERS

Troff provides a host of commands for placing characters of any size at any place. You can use them to draw special characters or to tune your output for a particular appearance. Most of these commands are straightforward, but they can be cryptic to read and tough to type correctly.

Although the **eqn** preprocessor described in Chapter 6 is the preferred tool for typesetting mathematical equations, you may elect to produce such rudimentary notations as subscripts and superscripts “by hand,” using **troff**’s half-line local motions: `\u` and `\d`. To go back up the page half a point-size, insert a `\u` at the desired place; to go down, insert a `\d`. `\u` (and `\d` should always be used in pairs, as explained below). Thus the input

```
Area = \(*pr\u2\d
```

produces the following output.

```
Area =  $\pi r^2$ 
```

To make the “2” smaller, bracket it with `\s-2...\s0`. Since `\u` and `\d` refer to the current point size, be sure to put *both* requests either inside or outside the size changes. If one is inside and one outside, you will get an unbalanced vertical motion.

Sometimes, the space given by `\u` and `\d` isn’t the right amount. The `\v` command can be used to request an arbitrary amount of vertical motion. The in-line command

```
\v'(amount)'
```

causes motion up or down the page by the amount specified in “(amount)”. For example, to move the “P” down, we used

```
.in +0.6i (move paragraph in)
.ll -0.3i (shorten lines)
.ti -0.3i (move P back)
\v'2'\s24\s0\v'-2'ater noster qui est
in caelis . . .
```

A minus sign causes upward motion, while no sign or a plus sign means

down the page. Thus, `\v' -2'` causes an upward vertical motion of two line spaces.

There are many other ways to specify the amount of motion:

```
\v'0.1i'
\v'3p'
\v'-0.5m'
```

are all legal, as are several other constructs. Notice that the scale specifier **i** or **p** or **m** goes inside the quotes. Any matching characters can be used in place of the quotes; this is also true of all other **troff** requests described in this section.

Since **troff** does not take within-the-line vertical motions into account when figuring out where it is on the page, output lines can have unexpected positions if the left and right ends aren't at the same vertical position. Thus, `\v`, like `\u` and `\d`, should always balance upward vertical motion in a line with the same amount in the downward direction.

Arbitrary horizontal motions are also available: `\h` is analogous to `\v`, except that the default scale factor is ems instead of line spaces. As an example,

```
\h'-0.1i'
```

causes a backward motion of a tenth of an inch. As a practical matter, consider printing the mathematical symbol “>>”. The default spacing is too wide, so **eqn** replaces this by

```
>\h'-0.3m'>
```

to produce >>.

Frequently `\h` is used with the “width function” `\w` to generate motions equal to the width of some character string. The construction

```
\w'string'
```

is a number equal to the width of “string” in machine units (1/432 inch). All **troff** computations are ultimately done in these units. To move horizontally the width of an “x”, we can say

```
\h'\w'x'u'
```

As we mentioned above, the default scale factor for all horizontal dimensions is **m** (ems), so here we must have the **u** to specify machine units. Otherwise the motion produced will be far too large. **Troff** is quite happy with the nested quotes so long as you don't leave any out.

As a live example of this kind of construction, all of the command names in the text, like **.sp**, were done by overstriking with a slight offset. The commands for **.sp** are

```
.sp\h'-\w'.sp'u'\h'1u'.sp
```

That is, put out “.sp”, move left by the width of “.sp”, move right 1

unit, and print “.sp” again.

There are also several special-purpose **troff** commands for local motion. We have already seen `\O`, which is an unpaddable white space of the same width as a digit. “Unpaddable” means that it will never be widened or split across a line by **troff**'s justification and filling process. There is also `\` (backslash space), which is an unpaddable character the width of a space, `\|`, which is half that width, `\^`, which is one quarter of the width of a space, and `\&`, which has zero width. (This last one is useful, for example, in entering a text line which would otherwise begin with a “.” and, therefore, look to **troff** like a request. It's a construction we used a lot in writing this chapter.)

The command `\o`, used like

```
\o'set of characters'
```

causes (up to 9) characters to be overstruck, centered on the widest. This is nice for accents, as in:

```
syst\o"e\(\ga"me t\o"e\(\aa"l\o"e\(\aa"phonique
```

which makes:

```
système téléphonique
```

The accents are `\(ga` and `\(aa`, or `\'` and `\'`; remember that each is just one character to **troff**.

You can make your own overstrikes with another special convention, `\z`, the zero-motion command. `\zx` suppresses the normal horizontal motion after printing the single character `x`, so another character can be laid on top of it. Although sizes can be changed within `\o`, it centers the characters on the widest, and there can be no horizontal or vertical motions, so `\z` may be the only way to get what you want. As an example, an extra-heavy semicolon that looks like

```
; instead of ; or ;
```

can be constructed with a big comma and a big period above it:

```
\s+6\z,\v'-0.25m'.\v'0.25m'\s0
```

A more ornate overstrike is given by the bracketing function `\b`, which piles up characters vertically, centered on the current baseline. Thus we can get big brackets, constructing them with piled-up smaller pieces:

```
{ [ x ] }
```

by typing in this:

```
.sp
\b' \(\t\(\l\(\l\(\l' \b' \(\l\(\l' x \b' \(\r\(\r' \b' \(\r\(\r\(\r\(\r'
```

Troff also provides a convenient facility for drawing horizontal and vertical lines of arbitrary length with arbitrary characters. The input

```
\l' 1i'
```

draws a line one inch long, like this: _____ . The length can be followed by the character to use if the `_` isn't appropriate; `\l' 0.5i.'` draws a half-inch line of dots: The construction `\L` is entirely analogous, except that it draws a vertical line instead of a horizontal one.

1.7 STRINGS

Obviously, if a document contains a large number of occurrences of an acute accent over the letter “e”, typing `\o”e\’` for each `é` would be a nuisance.

Fortunately, **troff** provides a way for you to store an arbitrary collection of text in a “string”, and thereafter use the string name as a shorthand for its contents. Strings are one of several **troff** mechanisms whose judicious use lets you type a document with less effort and organize it so that extensive format changes can be made with a few editing changes.

When **troff** processes your input file, a reference to a pre-defined string is replaced by the appropriate text. Strings are defined with the command `.ds`. The line:

```
.ds e \o”e\’ ”
```

defines the string `e` to have the value `\o”e\’` .

String names may be either one or two characters long, and are called by `*x` for one character names or `*(xy)` for two character names. Thus to get `téléphone`, given the definition of the string `e` as above, we can say `t*el*ephone`.

If a string must begin with blanks, define it as

```
.ds xx ” text
```

The double quote signals the beginning of the definition. There is no trailing quote, since the end of the line terminates the string.

A string may actually be several lines long; if **troff** encounters a `\` at the end of any line, it throws the `\` away and appends the next line to the current one. So you can make a long string simply by ending each line but the last with a backslash:

```
.ds xx this \
is a very \
long string
```

Strings may be defined in terms of other strings, or even in terms of themselves. We will discuss some of these possibilities later.

1.8 INTRODUCTION TO MACROS

In its simplest form, a macro is just a shorthand notation quite similar to a string. Suppose you want every paragraph to start in exactly the same way — with a space and a temporary indent of two ems:

```
.sp
.ti +2m
```

To save typing, you might like to collapse these into one shorthand line, a **troff** “request” like

```
.PP
```

that would be treated by **troff** exactly as

```
.sp
.ti +2m
```

.PP is called a *macro*. The way you tell **troff** what **.PP** means is to define it with the **.de** command:

```
.de PP
.sp
.ti +2m
..
```

The first line names the macro (we used **.PP** for “paragraph,” and uppercase so it wouldn’t conflict with any name that **troff** might already know about). The last line **..** “marks the end of the definition. In between is a collection of requests. These are inserted whenever **troff** sees the macro called

```
.PP
```

A macro can contain any combination of text and formatting commands.

The definition of **.PP** has to precede its first use; undefined macros are simply ignored. Names are restricted to one or two characters.

Using macros for commonly occurring sequences of commands is critically important. Not only does it save typing, but it makes later changes much easier. Suppose you decide that the paragraph indent is too small, the vertical space is much too big, and Roman font should be forced. Instead of changing the whole document, you need only change the definition of **.PP** to something like

```
.de PP \” paragraph macro
.sp 2p
.ti +3m
.ft R
..
```

and the change takes effect everywhere you used **.PP**.

The comment delimiter `\` is a **troff** request that causes the rest of the line to be ignored. We use it here to add comments to the macro definition (a wise idea once definitions get complicated, and essential to ensure comprehension by other users.)

As another example of macros, consider these two which start and end a block of offset, unfilled text, like most of the examples in this paper:

```
.de BS \" start indented block
.sp
.nf
.in +0.3i
..
.de BE \" end indented block
.sp
.fi
.in -0.3i
..
```

Now we can surround text like

```
Copy to
John Doe
Richard Roberts
Stanley Smith
```

by the commands **.BS** and **.BE**, and it will come out as it did above. Notice that we indented by **.in +0.3i** instead of **.in 0.3i**. This way we can nest our uses of **.BS** and **.BE** to get blocks within blocks.

1.9 TITLES, PAGES, AND NUMBERING

Suppose you want a title at the top of each page, saying simply:

```
left top      center top      right top
```

To do this right, you need to say what the actual title is (easy); when to print it (easy enough); and what to do at and around the title line (harder). Taking these in reverse order, first we define a macro **.NP** (for “new page”) to process titles and the like at the end of one page and the beginning of the next:

```
.de NP
' bp
' sp 0.5i
.tl 'left top'center top'right top'
' sp 0.3i
..
```

To make sure we’re at the top of a page, we issue a “begin page” command **' bp**, which causes a skip to top-of-page (we’ll explain the **'** shortly). Then we space down half an inch, print the title (the use of **.tl** should be self explanatory; later we will discuss parameterizing the

titles), space another 0.3 inches, and we're done.

To ask for **.NP** at the bottom of each page, we have to say something like "when the text is within an inch of the bottom of the page, start the processing for a new page." This is done with a "when" command **.wh**:

```
.wh -1i NP
```

(No "." is used before NP; this is simply the name of a macro, not a macro call.) The minus sign means "measure up from the bottom of the page", so "-1i" means "one inch from the bottom".

The **.wh** command appears in the input outside the definition of **.NP**; typically the input would be

```
.de NP
...
..
.wh -1i NP
```

Now what happens? As text is actually being output, **troff** keeps track of its vertical position on the page, and after a line is printed within one inch from the bottom, the **.NP** macro is activated. (In the jargon, the **.wh** command sets a *trap* at the specified place, which is "sprung" when that point is passed.) The **.NP** macro causes a skip to the top of the next page (that's what the **' bp** was for), then prints the title with the appropriate margins.

Why **' bp** and **' sp** instead of **.bp** and **.sp**? The answer is that **.sp** and **.bp**, like several other commands, cause a *break* to take place. That is, all the input text collected but not yet printed is flushed out as soon as possible, and the next input line is guaranteed to start a new line of output. If we had used **.sp** or **.bp** in the **.NP** macro, it would force a break in the middle of the current output line whenever a new page was started. The effect would be to print the left-over part of that line at the top of the page, followed by the next input line on a new output line. Using **'** instead of **.** for a command tells **troff** that no break is to take place — the output line currently being filled should not be forced out before the space or new page.

The following commands cause a break:

```
.bp .br .ce .fi .nf .sp .in .ti
```

All others cause *no* break, regardless of whether you use a **.** or a **'**. If you really need a break, add a **.br** command at the appropriate place.

In some cases, you may find that if you cross a page boundary in an unexpected font or size, your titles come out in that size and font instead of the one you intended. Furthermore, the length of a title is independent of the current line length. Titles will come out at the default length of 6.5 inches unless you change the title length with the **.lt** command.

There are several ways to fix the problems of point sizes and fonts in titles. For the simplest applications, we can change `.NP` to set the proper size and font for the title, then restore the previous values, like this:

```
.de NP
' bp
' sp 0.5i
.ft R \” set title font to roman
.ps 10 \” and size to 10 point
.lt 6i \” and length to 6 inches
.tl 'left'center'right'
.ps \” revert to previous size
.ft P \” and to previous font
' sp 0.3i
..
```

This version of `.NP` does not work if the fields in the `.tl` command contain size or font changes. To cope with that, `troff` provides an “environment” mechanism, which we will discuss in a later section.

1.9.1 Page Numbers

Output page numbers are computed automatically as each page is produced (starting at 1), but no numbers are printed unless you ask for them explicitly. To get page numbers printed, include the character `%` in the `.tl` line at the position where you want the number to appear. For example,

```
.tl ”- % -”
```

centers the page number inside hyphens. You can set the page number at any time with either `.bp n`, which immediately starts a new page numbered `n`, or with `.pn n`, which sets the page number for the next page but doesn't cause a skip to the new page. Again, `.bp +n` sets the page number to `n` more than its current value; `.bp` means `.bp +1`.

1.10 NUMBER REGISTERS AND ARITHMETIC

`Troff` has a facility for doing arithmetic, and for defining and using variables with numeric values, called number registers. Number registers, like strings and macros, can be useful in setting up a document so it is easy to change later. And, of course, they serve for any sort of arithmetic computation.

Like strings, number registers have one- or two-character names. They are set by the `.nr` command, and are referenced anywhere by `\nx` (one character name) or `\n(xy)` (two character name).

`Troff` maintains many pre-defined number registers, among them `%` for the current page number; `nl` for the current vertical position on the page; `dy`, `mo` and `yr` for the current day, month, and year; and `.s` and `.f` for

the current size and font. (The font is a number from 1 to 4.) Any of these can be used in computations like any other register, but some, like `.s` and `.f`, cannot be changed with `.nr`.

As an example of the use of number registers, in the `ms` macro package, most significant parameters are defined in terms of the values of a handful of number registers. These include the point size for text, the vertical spacing, and the line and title lengths. To set the point size and vertical spacing for the following paragraphs, for example, you may say

```
.nr PS 9
.nr VS 11
```

The `-ms` paragraph macro `.PP` is defined (roughly) as follows:

```
.de PP
.ps \\n(PS \ " reset size
.vs \\n(VSp \ " spacing
.ft R \ " font
.sp 0.5v \ " half a line
.ti +3m
..
```

This sets the font to Roman and the point size and line spacing to whatever values are stored in the number registers `PS` and `VS`.

Why are there two backslashes? This is the eternal problem of how to quote a quote. When `troff` originally reads the macro definition, it peels off the first backslash. To ensure that another is left in the definition when the macro is used, we have to put two backslashes in the definition. If only one backslash is used, `troff` will not get the correct message, and point size and vertical spacing will be unaffected when the macro is invoked.

Protecting by an extra layer of backslashes is only needed for `\n`, `*`, `\$` (which we haven't come to yet), and `\` itself. Things like `\s`, `\f`, `\h`, `\v`, and so on do not need an extra backslash, since they are converted by `troff` to an internal code immediately upon being seen.

Arithmetic expressions can appear anywhere that a number is expected. As a trivial example,

```
.nr PS \\n(PS-2
```

decrements `PS` by 2. Expressions can use the arithmetic operators `+`, `-`, `*`, `/`, `%` (mod), the relational operators `>`, `>=`, `<`, `<=`, `=`, and `!=` (not equal), and parentheses.

Although the arithmetic we have done so far has been straightforward, things can get somewhat tricky. First, number registers hold only integers. `Troff` arithmetic uses truncating integer division. Second, in the absence of parentheses, evaluation is done left-to-right without any operator precedence (including relational operators). For example,

`7*-4+3/13`

becomes “-1”. Number registers can occur anywhere in an expression, and so can scale indicators like **p**, **i**, **m**, and so on (but no spaces). Although integer division causes truncation, each number and its scale indicator is converted to machine units (1/432 inch) before any arithmetic is done, so `1i/2u` evaluates to 0.5i correctly.

The scale indicator **u** often has to appear when you wouldn’t expect it — in particular, when arithmetic is being done in a context that implies horizontal or vertical dimensions. For example,

`.ll 7/2i`

would seem to indicate a line length of 3.5 inches. But since the default units for horizontal parameters like `.ll` are ems, the above expression is read as “7 ems / 2 inches”, which when translated into machine units, becomes zero. A change to

`.ll 7i/2`

is still no good; the “2” is “2 ems”, so “7i/2” is small, although not zero. You *must* use

`.ll 7i/2u`

So again, the safest practice is to attach a scale indicator to all numbers, even constants.

For arithmetic done within a `.nr` command, there is no implication of horizontal or vertical dimension, so the default units are “units”, and `7i/2` and `7i/2u` mean the same thing. Thus

```
.nr ll 7i/2
.ll \\n(llu
```

does just what you want, so long as you don’t forget the **u** on the `.ll` command.

1.11 MACROS WITH ARGUMENTS

Troff allows you to define macros that can change from one use to the next according to parameters supplied as arguments to the macro. To make this work, you need two things: first, when you define the macro, you have to indicate that some parts of it will be provided as arguments when the macro is called. Then, when the macro is called, you have to provide actual arguments to be plugged into the definition.

Let us illustrate by defining a macro `.SM` that will print its argument two points smaller than the surrounding text. That is, the macro call

`.SM TROFF`

will produce TROFF

The definition of **.SM** is

```
.de SM
\s-2\\$1\s+2
..
```

Within a macro definition, the symbol `\\$n` refers to the *n*th argument that the macro was called with. Thus `\\$1` is the string to be placed in a smaller point size when **.SM** is called.

As a slightly more complicated version, the following definition of **.SM** permits optional second and third arguments that will be printed in the normal size:

```
.de SM
\\$3\s-2\\$1\s+2\\$2
..
```

Arguments not provided when the macro is called are treated as empty, so

```
.SM TROFF ),
```

produces TROFF), while

```
.SM TROFF ). (
```

produces (TROFF). It is convenient to reverse the order of arguments because trailing punctuation is much more common than leading.

The number of arguments that a macro was called with is available in number register `.$`.

The following macro **.BD** is the one used to make the “bold roman” we have been using for **troff** request names in text. It combines horizontal motions, width computations, and argument rearrangement.

```
.de BD
\&\\$3\f1\\$1\h'-\w'\\$1'u+1u'\\$1\fP\\$2
..
```

As we previously mentioned, the `\h` and `\w` need no extra backslash. The `\&` is there in case the argument begins with a period.

Two backslashes are needed with the `\\$n` commands, though, to protect one of them when the macro is being defined. Perhaps a second example will make this clearer. Consider a macro called **.SH** which might be used to produce section headings rather like those in this chapter, with the sections numbered automatically, and the title in bold in a smaller size. The use is

```
.SH "Section title . . ."
```

(If an argument to a macro is to contain blanks, it must be surrounded by double quotes, unlike a string, where only one leading quote is permitted.)

Here is the definition of the **.SH** macro:

```
.nr SH 0 \ " initialize section number
.de SH
.sp 0.3i
.ft B
.nr SH \n(SH+1 \ " increment number
.ps \n(PS-1 \ " decrease PS
\n(SH. \ $1 \ " number. title
.ps \n(PS \ " restore PS
.sp 0.3i
.ft R
..
```

The section number is kept in number register SH, which is incremented each time just before it is used.

Note: A number register may have the same name as a macro. A string may not.

We used `\n(SH` instead of `\nSH` and `\n(PS` instead of `\nPS`. If we had used `\nSH`, we would get the value of the register at the time the macro was defined, not at the time it was used. Similarly, by using `\n(PS`, we get the point size at the time the macro is called.

As an example that does not involve numbers, recall our **.NP** macro which had a

```
.tl 'left'center'right'
```

We could make these into parameters by using instead

```
.tl '\*(LT'\*(CT'\*(RT'
```

so the title comes from three strings called LT, CT, and RT. If these are empty, then the title will be a blank line. Normally CT would be set with something like

```
.ds CT - % -
```

to give just the page number between hyphens (as on the top of this page), but a user could supply private definitions for any of the strings.

1.12 CONDITIONALS

Suppose you want the **.SH** macro to leave two extra inches of space just before Section 1, but nowhere else. The cleanest way to do that is to test inside the **.SH** macro whether the section number is 1, and add some space if it is. The **.if** command provides a conditional test that we can add just before the heading line is output:

```
.if \n(SH=1 .sp 2i \ " first section only
```

The condition after the `.if` can be any arithmetic or logical expression. If the condition is logically true, or arithmetically greater than zero, then the rest of the line is treated as if it were text — here it is a command. If the condition is false, or zero or negative, the rest of the line is skipped.

It is possible to do more than one command if a condition is true. Suppose several operations are to be done before Section 1. One possibility is to define a macro `.S1` and invoke it if we are about to do Section 1 (as determined by an `.if`).

```
.de S1
--- processing for Section 1 ---
..
.de SH
...
.if \\n(SH=1 .S1
...
..
```

An alternate way is to use the extended form of the `.if`, like this:

```
.if \\n(SH=1 \{--- processing
for Section 1 ----\}
```

The braces `\{` and `\}` must occur in the positions shown or you will get unexpected extra lines in your output. **Troff** also provides an “if-else” construction, which is treated in more detail in Chapter 2 of this section.

A condition can be negated by preceding it with `!`; we get the same effect as above (but less clearly) by using

```
.if !\\n(SH>1 .S1
```

There are a handful of other conditions that can be tested with `.if`. For example, is the current page even or odd?

```
.if e .tl "even page title"
.if o .tl "odd page title"
```

gives facing pages different titles when used inside an appropriate new page macro.

Two other conditions are `t` and `n`, which tell you whether the formatter is **troff** or **nroff**.

```
.if t troff stuff ...
.if n nroff stuff ...
```

Finally, string comparisons may be made in an `.if`:

```
.if 'string1'string2' stuff
```

does “stuff” if *string1* is the same as *string2*. The character separating the strings can be anything reasonable that is not contained in either string. The strings themselves can reference strings with `*`, arguments

with `\$`, and so on.

1.13 ENVIRONMENTS

As we mentioned, there is a potential problem when going across a page boundary: parameters like size and font for a page title may well be different from those in effect in the text when the page boundary occurs. **Troff** provides a very general way to deal with this and similar situations. There are three “environments”, each of which has independently settable versions of many of the parameters associated with processing, including size, font, line and title lengths, fill/no-fill mode, tab stops, and even partially collected lines. Thus, the titling problem may be readily solved by processing the main text in one environment and titles in a separate one with its own suitable parameters.

The command `.ev n` shifts to environment `n`; `n` must be 0, 1 or 2. The command `.ev` with no argument returns to the previous environment. Environment names are maintained in a stack, so calls for different environments may be nested and unwound consistently.

Suppose we say that the main text is processed in environment 0, which is where **troff** begins by default. Then we can modify the new page macro `.NP` to process titles in environment 1 like this:

```
.de NP
.ev 1 \” shift to new environment
.lt 6i \” set parameters here
.ft R
.ps 10
... any other processing ...
.ev \” return to previous environment
..
```

It is also possible to initialize the parameters for an environment outside the `.NP` macro, but the version shown keeps all the processing in one place, making it easier to understand and change.

1.14 DIVERSIONS

There are numerous occasions in page layout when it is necessary to store some text for a period of time without actually printing it. Footnotes are the most obvious example: the text of the footnote usually appears in the input well before the place on the page where it is to be printed is reached. In fact, the place where it is output normally depends on how big it is, which implies that there must be a way to process the footnote at least enough to decide its size without printing it.

Troff provides a mechanism called a diversion for doing this processing. Any part of the output may be diverted into a macro instead of being printed, and then at some convenient time, the macro may be put back into the input.

The command **.di xy** begins a diversion; all subsequent output is collected into the macro **xy** until the command **.di** with no arguments is encountered. This terminates the diversion. The processed text is available at any time thereafter, simply by giving the command

```
.xy
```

The vertical size of the last finished diversion is contained in the built-in number register **dn**.

As a simple example, suppose we want to implement a “keep-release” operation, so that text between the commands **.KS** and **.KE** will not be split across a page boundary (as for a figure or table). Clearly, when a **.KS** is encountered, we have to begin diverting the output so we can find out how big it is. Then when a **.KE** is seen, we decide whether the diverted text will fit on the current page, and print it either there if it fits, or at the top of the next page if it doesn't. So:

```
.de KS \” start keep
.br   \” start fresh line
.ev 1 \” collect in new environment
.fi   \” make it filled text
.di XX \” collect in XX
..

.de KE \” end keep
.br   \” get last partial line
.di   \” end diversion
.if   \\n(dn>=\\n(.t.bp \” bp if doesn't fit
.nf   \” bring it back in no-fill
.XX   \” text
.ev   \” return to normal environment
..
```

Recall that number register **nl** holds the current position on the output page. Since output was being diverted, this remains at its value when the diversion started. **dn** is the amount of text in the diversion; **t** (another built-in register) is the distance to the next trap, which we assume is at the bottom margin of the page. If the diversion is large enough to go past the trap, the **.if** is satisfied, and a **.bp** is issued. In either case, the diverted output is then brought back with **.XX**. It is essential to bring it back in no-fill mode so **troff** will do no further processing on it.

This is not the most general keep-release, nor is it robust in the face of all conceivable inputs, but it would require more space than we have here to describe a more general mechanism. The **-ms**, **-me**, and **-mm** macro packages all have keep mechanisms. They are described in chapters 3, 4, and 5 respectively.

C

C

C

C

C

Chapter 2: The troff Reference Manual

2.1 INTRODUCTION

Nroff and **troff** are text formatting programs. **Nroff** formats text for a variety of dot-matrix and letter-quality printers (as well as for DM windows and most CRTs), and **troff** does the same thing for the Graphic Systems CAT phototypesetter. Both **nroff** and **troff** accept lines of text interspersed with lines of format control information and output formatted text, usually for a specific output device.

Note: This chapter is a reference manual for **nroff** and **troff**. Of necessity, it is technical in nature, and may tell you a good deal more than you really need to know. For a more measured introduction to the subject, see Chapter 1 of this section.

Nroff and **troff** have similar input file requirements. It is almost always possible to prepare input acceptable to both. Conditional input requests let you embed input expressly destined for either program. When we refer to **troff** in this chapter, we mean “troff and/or nroff.” Where a feature is unique to one or the other we will say so. Otherwise, you may safely assume that they behave identically.

2.1.1 Usage

The following line is the model for lines that invoke **nroff** from either a Bourne or a C shell.

```
nroff options file(s)
```

Similarly, the following line is the model for lines that invoke **troff**.

```
troff options file(s)
```

where *options* represents any of a number of optional arguments and *file(s)* represents the file (or list of files separated by spaces) containing the document(s) to be formatted. An argument consisting of a single minus (–) is taken to be a file name corresponding to the standard input. If no file names are given, input is taken from the standard input. The options must appear before the file(s), but may appear in any order.

Option	Effect
-olist	Print only the pages whose page numbers appear in the <i>list</i> of numbers and/or number ranges separated by commas. A number range has the form <i>N-M</i> and means pages <i>N</i> through <i>M</i> . An initial <i>-N</i> means from the beginning to page <i>N</i> ; and a final <i>N-</i> means from <i>N</i> to the end.

- n***N* Number first generated page *N*.
- s***N* Stop every *N* pages. **Nroff** will halt every *N* pages (default *N*=1) to allow paper loading or changing, and will resume upon receipt of a newline. **Troff** will stop the CAT phototypesetter every *N* pages, produce a trailer to allow changing cassettes, and will resume after the phototypesetter's START button is pressed.
- m***name* Prepends the macro file
 /*usr/lib/tmac.name*
 to the input *files*.
- r***aN* Number register *a* (one-character) is set to *N*. May be repeated to set multiple *a*'s.
- i** Read standard input after the end of the specified *file(s)*.
- q** Invoke the simultaneous input-output mode of the **.rd** request.

Nroff Only

- | Option | Effect |
|-----------------------|---|
| -T <i>name</i> | Specifies the name of the output terminal type. Must be a type included in the directory <i>/usr/lib/tab</i> . See your System Administrator for a list of the terminal types available at your installation and their names. |
| -e | Produce equally-spaced words in adjusted lines, using full terminal resolution. |

Troff Only

- | Option | Effect |
|--------------------|--|
| -t | Direct output to standard output instead of to the phototypesetter. |
| -f | Refrain from feeding out paper and stopping phototypesetter at the end of the run. |
| -w | Wait until phototypesetter is available if it is currently busy. |
| -b | Report whether the phototypesetter is busy or available. Do not do text processing. |
| -a | Send a printable ASCII approximation of the results to standard output. |
| -p <i>N</i> | Print all characters in point size <i>N</i> while retaining all prescribed spacings and motions, to reduce phototypesetter elapsed time. |

Each option is invoked as a separate argument; for example,

```
nroff -o4,8-10 -T300S -mm file1 file2
```

requests formatting of pages 4, 8, 9, and 10 of a document contained in the files named *file1* and *file2*, specifies the output terminal as a DASI-300S, and invokes the macro package *tmac.m*, better known as the **-mm** macro package (see Chapter 5).

Various pre- and post-processors are available for use with **nroff** and **troff**. These include the equation preprocessors **eqn** and **neqn**, the table formatter **tbl**, and the **refer** bibliography macros. Chapters 6 and 7 of this section include more information about pre-processors. Normally, you connect **troff** with these pre-, and post-processors by means of pipes. For example, in

```
tbl files | eqn | troff -t options
```

the first | indicates the piping of **tbl**'s output to **eqn**'s input. The second indicates the piping of output to **troff**'s input.

2.1.2 Input File Format

A **troff** or **nroff** input file consists of text lines, which are destined to be printed, interspersed with formatter requests, which set parameters or otherwise control subsequent processing. Formatter requests normally begin with a . (period) or, in some cases, a ' (acute accent). The request character ' suppresses the break function induced by certain requests. A break usually forces output of any partially collected input line — something that you may need to avoid when defining a macro.

The request character is followed by a one- or two-character name that specifies either a basic request (one of the ones defined in this chapter) or a user-defined or macro that may invoke many requests. In general, we will try to differentiate between requests, which are defined in the **troff** program itself and cannot be altered by the user, and macros, which are defined by the user or obtained from a predefined macro package such as **-me** or **-mm**. The request character may be separated from the request/macro name by zero or more spaces and/or tabs. Request or macro names must be followed by either a space or a newline. Lines that begin with a request character but include an unrecognized name are ignored.

Various special functions may be introduced anywhere in the input by means of an escape character, normally the backslash.

\

For example, the function **\nR** causes the interpolation of the contents of the number register **R**. Number registers are discussed in detail in a later section.

2.1.3 Output Device Resolution

Troff resolves horizontal distances to 1/432'd inch. This number is derived from the CAT phototypesetter which has a horizontal resolution of 1/432 inch and a vertical resolution of 1/144 inch. **nroff** resolves to 240 units/inch, corresponding to the lowest common multiple of the horizontal and vertical resolutions of various printing terminals. **Troff** rounds horizontal/vertical numerical parameter input to the actual resolution of the CAT. **Nroff** similarly rounds numerical input to the actual resolution of the output device indicated by the **-T** option (default Model 37 Teletype).

2.1.4 Numerical Parameter Input

Troff accepts numerical input with the appended scale indicators shown in the following table, where *S* is the current type size in points, *V* is the current vertical line spacing in basic units, and *C* is a nominal character width in basic units.

Scale Indicator	Meaning	Number of basic units	
		troff	nroff
i	Inch	432	240
c	Centimeter	$432 \times 50 / 127$	$240 \times 50 / 127$
P	Pica = 1/6 inch	72	240/6
m	Em = <i>S</i> points	$6 \times S$	<i>C</i>
n	En = Em/2	$3 \times S$	<i>C</i> , same as <i>Em</i>
p	Point = 1/72 inch	6	240/72
u	Basic unit	1	1
v	Vertical line space	<i>V</i>	<i>V</i>
none	Default, see below		

In **nroff**, the em and the en are equal fixed-width character-sized increments. Common values are 1/10 and 1/12 inch. Actual character widths in **nroff** need not be all the same and constructed characters such as **->** (**->**) are often extra wide.

The default scaling is ems for the horizontally-oriented requests and functions **ll**, **in**, **ti**, **ta**, **lt**, **po**, **mc**, **\h**, and **\l**; *V*s for the vertically-oriented requests and functions **pl**, **wh**, **ch**, **dt**, **sp**, **sv**, **ne**, **rt**, **\v**, **\x**, and **\L**; **p** for the **vs** request; and **u** for the requests **nr**, **if**, and **ie**. All other requests ignore any scale indicators.

When a number register containing an appropriately scaled number is interpolated to provide numerical input, the unit scale indicator **u** may need to be appended to prevent an additional inappropriate default scaling. The number may be specified in decimal-fraction form but the parameter finally stored is rounded to an integer number of basic units.

The absolute position indicator **~** (tilde) may be prepended to a number *N* to generate the distance to the vertical or horizontal place *N*. For vertically-oriented requests and functions, **~ N** becomes the distance in basic units from the current vertical place on the page or in a diversion

to the the vertical place N . For all other requests and functions, $\sim N$ becomes the distance from the current horizontal place on the *input* line to the horizontal place N . For example,

.sp \sim 3.2c

will space in the required direction to a point 3.2 centimeters from the top of the page.

2.1.5 Numerical Expressions

Wherever numerical input is expected, you may use an expression involving parentheses, the arithmetic operators $+$, $-$, $/$, $*$, $\%$ (mod), and the logical operators $<$, $>$, $<=$, $>=$, $=$ (or $==$), $\&$ (and), and $:$ (or). Except where controlled by parentheses, these expressions are evaluated left-to-right. There is no operator precedence. In the case of certain requests, an initial $+$ or $-$ is stripped and interpreted as an increment or decrement indicator respectively. In the presence of default scaling, the desired scale indicator must be attached to every number in an expression for which the desired and default scaling differ. For example, if the number register x contains 2 and the current point size is 10, then

.ll (4.25i+\nxP+3)/2u

will set the line length to half the sum of 4.25 inches + 2 picas + 30 points.

2.1.6 Notational Conventions

In this chapter, we indicate numerical parameters in two ways: $\pm N$ means that the argument may take the forms N , $+N$, or $-N$ and that the corresponding effect is to set the affected parameter to N , to increment it by N , or to decrement it by N respectively. Plain N means that an initial algebraic sign is not an increment indicator, but merely the sign of N . Generally, unreasonable numerical input is either ignored or truncated to a reasonable value. For example, most requests expect to set parameters to non-negative values. Exceptions are **sp**, **wh**, **ch**, **nr**, and **if**. The requests **ps**, **ft**, **po**, **vs**, **ls**, **ll**, **in**, and **lt** restore the previous parameter value in the absence of an argument.

Single-character arguments are indicated by single lowercase letters. One- or two-character arguments are indicated by a pair of lower case letters. Character string arguments are indicated by multi-character mnemonics.

2.2 FONT AND CHARACTER SIZE CONTROL

2.2.1 Character Set

The **troff** character set consists of a standard "Times" family character set plus a Special Mathematical character set. Each set includes 102 characters. There is a table of these character sets at the end of this chapter. All ASCII characters are included, with some on the Special

Font. With three exceptions, the ASCII characters are input as themselves, and non-ASCII characters are input in the form $\backslash(xx$ where xx is a two-character name given in the table at the end of this chapter. The three ASCII exceptions are mapped as follows:

ASCII Input		Printed by troff	
Character	Name	Character	Name
'	acute accent	'	close quote
`	grave accent	'	open quote
-	minus	-	hyphen

The characters $'$, $`$, and $-$ may be input by \backslash' , $\backslash`$, and $\backslash-$ respectively or by using the names detailed in the table at the end of this chapter. The ASCII characters $@$, $\#$, $"$, $'$, $`$, $<$, $>$, \backslash , $\{$, $\}$, \sim , \hat , and $_$ exist only on the Special Font and are printed as a 1-em space if that font is not mounted.

Nroff understands the entire **troff** character set, but can usually print only ASCII characters, those additional characters available on the output device, characters that can be constructed by overstriking or other means, and those that can reasonably be mapped into other printable characters. The exact behavior is determined by a driving table prepared for each device. The following characters

'
`
-

print as themselves.

2.2.2 Fonts

The default mounted fonts are Times Roman (**R**), Times Italic (**I**), Times Bold (**B**), and the Special Mathematical Font (**S**) at font positions 1, 2, 3, and 4 respectively. The current font, initially Roman, may be changed (among the mounted fonts) by use of the **ft** request, or by imbedding at any desired point either $\backslash fx$, $\backslash f(xx$, or $\backslash fN$ where x and xx are the name of a mounted font and N is a numerical font position. It is not necessary to change to the Special font; characters on that font are handled automatically. A request for an unmounted font is ignored. **Troff** can be informed that a font is mounted by use of the **fp** request. See your System Administrator for a list of the fonts available at your site. In the subsequent discussion of font-related requests, F represents either a one-/two-character font name or the numerical font position, 1-4. The current font is available (as numerical position) in the read-only number register **.f**.

2.2.3 Character Size

Nroff ignores type size control. The CAT typesetter (and, hence, **troff**) supports the following character point sizes: 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36. This provides a size range of 1/12 inch to 1/2 inch. The **ps** request is used to change or restore the point size.

Alternatively, the point size may be changed by imbedding a `\sN` in running text to set the size to N , or a `\s±N` ($1 \leq N \leq 9$) to increment/decrement the size by N . The request `\s0` restores the previous size. If you request a point size that is between two valid sizes, **troff** will use the larger of the two. The current size is available in the `.s` register.

In this and all similar tables in this chapter, we use the following designations in the *Notes* column.

- B** Request normally causes a break.
- D** Mode or relevant parameters associated with current diversion level.
- E** Relevant parameters are a part of the current environment.
- O** Must stay in effect until logical output.
- P** Mode must be still or again in effect at the time of physical output.

The characters **v,p,m,u** are default scale indicators; if not specified, scale indicators are ignored.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.ps±N</code>	10 point	previous	E	Point size set to $\pm N$. Alternatively imbed <code>\sN</code> or <code>\s±N</code> . Any positive size value may be requested; if invalid, the next larger valid size will result, with a maximum of 36. A paired sequence $+N, -N$ will work because the previous requested value is also remembered. Ignored in nroff .
<code>.ssN</code>	12/36 em	ignored	E	Space-character size is set to $N/36$ ems. This size is the minimum word spacing in adjusted text. Ignored in nroff .
<code>.csFNM</code>	off	-	P	Constant character space (width) mode is set on for font F (if mounted); the width of every character will be taken to be $N/36$ ems. If M is absent, the em is that of the character's point size; if M is given, the em is M -points. All affected characters are centered in this space, including those with an actual width larger

				than this space. Special Font characters occurring while the current font is <i>F</i> are also so treated. If <i>N</i> is absent, the mode is turned off. The mode must be still or again in effect when the characters are physically printed. Ignored in nroff .
.bdFN	off	-	P	The characters in font <i>F</i> will be artificially emboldened by printing each one twice, separated by <i>N</i> -1 basic units. A reasonable value for <i>N</i> is 3 when the character size is in the vicinity of 10 points. If <i>N</i> is missing the embolden mode is turned off. The column heads for this table were printed with .bdI3 . The mode must be still or again in effect when the characters are physically printed. Ignored in nroff .
.bdSFN	off	-	P	The characters in the Special Font will be emboldened whenever the current font is <i>F</i> . The mode must be still or again in effect when the characters are physically printed.
.ftF	Roman	previous	E	Font changed to <i>F</i> . Alternatively, imbed \fF . The font name P is reserved to mean the previous font.
.fpNF	R,I,B,S	ignored	-	Font position. This is a statement that a font named <i>F</i> is mounted on position <i>N</i> (1-4). It is a fatal error if <i>F</i> is not known. The default mounting sequence assumed by troff is R, I, B, and S on positions 1, 2, 3, and 4.

2.3 PAGE CONTROL

Top and bottom margins are not automatically provided; it is conventional to define two macros and to set traps for them at vertical positions 0 (top) and $-N$ (*N* from the bottom). A pseudo-transition onto the first page occurs either when the first break occurs or when the first non-diverted text processing occurs. Arrangements for a trap to occur at the top of the first page must be completed before this transition. In the

following table, references to the current diversion mean that the mechanism being described works during both ordinary and diverted output (the former considered as the top diversion level).

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.pl$\pm N$	11 in	11 in	v	Page length set to $\pm N$. The internal limitation is about 75 inches in troff and about 136 inches in nroff . The current page length is available in the .p register.
.bp$\pm N$	$N=1$	-	B,v	Begin page. The current page is ejected and a new page is begun. If $\pm N$ is given, the new page number will be $\pm N$. The use of " ' " as request character (instead of ".") suppresses the break function. Also, see request ns .
.pn$\pm N$	$N=1$	ignored	-	Page number. The next page (when it occurs) will have the page number $\pm N$. A pn must occur before the initial pseudo-page transition to effect the page number of the first page. The current page number is in the % register.
.po$\pm N$	0; 26/27 i	previous	v	Page offset. Values separated by ";" are for nroff and troff respectively. The current <i>left margin</i> is set to $\pm N$. The troff initial value provides about 1 inch of paper margin including the physical typesetter margin of 1/27th inch. In troff the maximum (line length) + (page offset) is about 7.54 inches. The current page offset is available in the .o register.
.neN	-	$N=1$ V	D,v	Need N vertical space. If the distance, D , to the next trap position is less than N , a forward vertical space of size D occurs, which will spring the trap. If no traps are left the page, D is the distance to the bottom of the

				page. If $D < V$, another line could still be output and spring the trap. In a diversion, D is the distance to the <i>diversion trap</i> , if any, or is very large.
.mk R	none	internal	D	Mark the <i>current</i> vertical place in an internal register (both associated with the current diversion level), or in register R , if given. See rt request.
.rt $\pm N$	none	internal	D,v	Return (move up) to a marked vertical place in the current diversion. If $\pm N$ (w.r.t. current place) is given, the place is $\pm N$ from the top of the page or diversion or, if N is absent, to a place marked by a previous mk . Note that the sp request may be used in all cases instead of rt by spacing to the absolute place stored in an explicit register; e. g. using the sequence .mk $ R$sp $ \backslash nRu$.

2.4 TEXT FILLING AND ADJUSTING

2.4.1 Filling and Adjusting

Normally, words are collected from input text lines and assembled into an output text line until some word doesn't fit. An attempt is then made to hyphenate the word in an effort to assemble a part of it into the output line. The spaces between the words on the output line are then increased to spread out the line to the current line length minus any current indent.

A word is any string of characters delimited by the space character or the beginning/end of the input line. Any adjacent pair of words that must be kept together (neither split across output lines nor spread apart in the adjustment process) can be tied together by separating them with the unpaddingable space character “\ ” (backslash-space). The adjusted word spacings are uniform in **troff**. The minimum interword spacing can be controlled with the **ss** request. In **nroff**, they are normally nonuniform because of quantization to character-size spaces; however, the command line option **-e** causes uniform spacing with full output device resolution. Filling, adjustment, and hyphenation can all be prevented or controlled. The text length on the last line output is available in the **.n** register, and text baseline position on the page for this line is in the **nl** register. The text base-line high-water mark (lowest place) on the current page is in the **.h** register.

An input text line ending with ., ?, or ! is taken to be the end of a sentence, and an additional space character is automatically provided during filling. Multiple inter-word space characters found in the input are retained, except for trailing spaces. An initial space also causes a break.

When filling is in effect, a `\p` may be imbedded or attached to a word to cause a break at the end of the word and have the resulting output line spread out to fill the current line length.

If you need to begin a text input line with a request character (e.g., a dot), preface the line with the non-printing, zero-width filler character `\&`. You may also achieve the same effect by specifying output translation of some convenient character into the request character using `tr`.

2.4.2 Interrupted Text

The copying of a input line in nofill mode can be interrupted by terminating the partial line with a `\c` (concealed newline). The next input text line encountered will be considered to be a continuation of the same line of input text. Similarly, a word within filled text may be interrupted by terminating the word (and line) with `\c`. The next encountered text will be taken as a continuation of the interrupted word. If the intervening control lines cause a break, any partial line will be forced out along with any partial word.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.br</code>	-	-	B	Break. The filling of the line currently being collected is stopped and the line is output without adjustment. Text lines beginning with space characters and empty text lines (blank lines) also cause a break.
<code>.fi</code>	fill on	-	B,E	Fill subsequent output lines. The register <code>.u</code> is 1 in fill mode and 0 in nofill mode.
<code>.nf</code>	fill on	-	B,E	Nofill. Subsequent output lines are neither filled nor adjusted. Input text lines are copied directly to output lines without regard for the current line length.
<code>.adc</code>	adj. l,r	adjust	E	Line adjustment is begun. If fill mode is not on, adjustment will be deferred until fill mode is back on. If the type indicator <code>c</code> is present, the adjustment type is changed as shown in the following table.

Indicator	Adjustment
l	adjust left margin only
r	adjust right margin only
c	center
b or n	adjust both margins
absent	unchanged

- .na** adjust - E Noadjust. Adjustment is turned off; the right margin will be ragged. The adjustment type for **ad** is not changed. Output line filling still occurs if fill mode is on.
- .ce** *N* off *N=1* B,ECenter the next *N* input text lines within the current (line-length minus indent). If *N=0*, any residual count is cleared. A break occurs after each of the *N* input lines. If the input line is too long, it will be left-adjusted.

2.5 VERTICAL SPACING

2.5.1 Baseline Spacing

The vertical spacing (*V*) between the baselines of successive output lines can be set using the **vs** request with a resolution of $1/144$ inch = $1/2$ point in **troff**, and to the output device resolution in **nroff**. *V* must be large enough to accommodate the character sizes on the affected output lines. For the common type sizes (9-12 points), usual typesetting practice is to set *V* to 2 points greater than the point size; **troff** default is 10-point type on a 12-point spacing. (This document is set in 12-point type on 13.5-point spacing). The current *V* is available in the **.v** register. Multiple-*V* line separation (e. g. double spacing) may be requested with **ls**.

2.5.2 Extra Line Space

If a word contains a vertically tall construct requiring the output line containing it to have extra vertical space before and/or after it, the extra-line space function $\backslash x' N'$ can be imbedded in or attached to that word. In this and other functions having a pair of delimiters around their parameter (here $'$), the delimiter choice is arbitrary, except that it can't look like the continuation of a number expression for *N*. If *N* is negative, the output line containing the word will be preceded by *N* extra vertical space; if *N* is positive, the output line containing the word will be followed by *N* extra vertical space. If successive requests for extra space apply to the same line, the maximum values are used. The most recently utilized post-line extra line-space is available in the **.a** register.

2.5.3 Blocks of Vertical Space

A block of vertical space is ordinarily requested using **sp**, which honors the no-space mode and which does not space past a trap. A contiguous block of vertical space may be reserved using **sv**.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.vs <i>N</i>	12pts	previous	E, p	Set vertical baseline spacing size <i>V</i> . Transient extra vertical space available with \x'N' (see above). Nroff uses a default of 1/6th inch.
.ls <i>N</i>	<i>N</i> =1	previous	E	<i>Line</i> spacing set to $\pm N$. <i>N</i> -1 <i>V</i> s (<i>blank lines</i>) are appended to each output text line. Appended blank lines are omitted, if the text or previous appended blank line reached a trap position.
.sp <i>N</i>	-	<i>N</i> =1 <i>V</i>	B, v	Space vertically in <i>either</i> direction. If <i>N</i> is negative, the motion is backward (up) and is limited to the distance to the top of the page. Forward (down) motion is truncated to the distance to the nearest trap. If the no-space mode is on, no spacing occurs (see ns , and rs below).
.sv <i>N</i>	-	<i>N</i> =1 <i>V</i>	v	Save a contiguous vertical block of size <i>N</i> . If the distance to the next trap is greater than <i>N</i> , <i>N</i> vertical space is output. No-space mode has no effect. If this distance is less than <i>N</i> , no vertical space is immediately output, but <i>N</i> is remembered for later output (see os). Subsequent sv requests will overwrite any still remembered <i>N</i> .
.os	-	-	-	Output saved vertical space. No-space mode has <i>no</i> effect. Used to finally output a block of vertical space requested by an earlier sv request.
.ns	space	-	D	No-space mode turned on. When on, the no-space mode inhibits sp requests and bp requests without

				a next page number. The no-space mode is turned off when a line of output occurs, or with rs .
.rs	space	-	D	Restore spacing. The no-space mode is turned off.
blankline	-	-	B	Causes a break and output of a blank line exactly like sp 1 .

2.6 LINE LENGTH AND INDENTING

The maximum line length for fill mode may be set with **ll**. The indent may be set with **in**; an indent applicable to only the next output line may be set with **ti**. The line length includes indent space but not page offset space. The line-length minus the indent is the basis for centering with **ce**. The effect of **ll**, **in**, or **ti** is delayed, if a partially collected line exists, until after that line is output. In fill mode the length of text on an output line is less than or equal to the line length minus the indent. The current line length and indent are available in registers **.l** and **.i** respectively. The length of three-part titles produced by **tl** is independently set by **lt**.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.ll $\pm N$	6.5 in	previous	E, m	Line length is set to $\pm N$. In troff the maximum (line length) + (page offset) is about 7.54 inches.
.in $\pm N$	$N=0$	previous	B,E, m	Indent is set to $\pm N$. The indent is prepended to each output line.
.ti $\pm N$	-	ignored	B,E, m	Temporary indent. The next output text line will be indented a distance $\pm N$ with respect to the current indent. The resulting total indent may not be negative. The current indent is not changed.

2.7 MACROS, STRINGS, DIVERSIONS, TRAPS

2.7.1 Macros and Strings

A macro is a named set of arbitrary lines that may be invoked by name or with a trap. A string is a named string of characters, not including a newline character, that may be interpolated by name at any point. Request, macro, and string names share the same name list. Macro and string names may be one or two characters long and may usurp previously defined request, macro, or string names. Any of these entities may be renamed with **rn** or removed with **rm**. Macros are created by **de** and

di, and appended to by **am** and **da**; **di** and **da** cause normal output to be stored in a macro. Strings are created by **ds** and appended to by **as**. A macro is invoked in the same way as a request; a control line beginning **.xx** will interpolate the contents of macro **xx**. The remainder of the line may contain up to nine arguments. The strings **x** and **xx** are interpolated at any desired point with ***x** and ***(xx** respectively. String references and macro invocations may be nested.

2.7.2 Copy Mode Input Interpretation

During the definition and extension of strings and macros, the input is read in copy mode. That is to say, input is copied without interpretation, except for the following special cases.

- The contents of number registers indicated by **\n** are interpolated.
- Strings indicated by ***** are interpolated.
- Arguments indicated by **\\$** are interpolated.
- Concealed newlines indicated by **\(newline)** are eliminated.
- Comments indicated by **\"** are eliminated.
- **\t** and **\a** are interpreted as ASCII horizontal tab and SOH respectively.
- **** is interpreted as **"\"**.
- **\.** is interpreted as **"."**.

These interpretations can be suppressed by prepending a ****. For example, since **** maps into a ****, **\\n** will copy as **\n** which will be interpreted as a number register indicator when the macro or string is reread.

2.7.3 Arguments

When a macro is invoked by name, **troff** assumes that the remainder of the line contains arguments to that macro. Up to nine arguments are allowed. Arguments are separated by the space character. Arguments that contain spaces must be surrounded by double-quotes. Pairs of double-quotes may be imbedded in double-quoted arguments to represent a single double-quote. If the desired arguments won't fit on a line, a concealed newline may be used to continue on the next line.

When a macro is invoked, the input level is pushed down and any arguments available at the previous level become unavailable until the macro is completely read and the previous level is restored. A macro's own arguments can be interpolated at any point within the macro with **\\$N**, which interpolates the *N*th argument ($1 \leq N \leq 9$). If an invoked argument doesn't exist, a null string results. For example, the macro **xx** may be defined by

```
.de xx \ "begin definition
Today is \\$1 the \\$2.
.. \ "end definition
```

and called by

```
.xx Monday 14th
```

to produce the text

```
Today is Monday the 14th.
```

Note that the `\$` was concealed in the definition with a prepended `\`. The number of currently available arguments is in the `.$` register.

Because string referencing is implemented as an input-level push down, no arguments are available from within a string. No arguments are available within a trap-invoked macro.

Arguments are copied in copy mode onto a stack where they are available for reference. The mechanism does not allow an argument to contain a direct reference to a long string (interpolated at copy time) and it is advisable to conceal string references (with an extra `\`) to delay interpolation until argument reference time.

2.7.4 Diversions

Processed output may be diverted into a macro for purposes such as footnote processing or determining the horizontal and vertical size of some text for conditional changing of pages or columns. A single diversion trap may be set at a specified vertical position. The number registers `dn` and `dl` respectively contain the vertical and horizontal size of the most recently ended diversion. Processed text that is diverted into a macro retains the vertical size of each of its lines when reread in `nofill` mode regardless of the current `V`. Constant-spaced (`cs`) or emboldened (`bd`) text that is diverted can be reread correctly only if these modes are again or still in effect at reread time. One way to do this is to imbed in the diversion the appropriate `cs` or `bd` requests with the transparent mechanism described later in this chapter.

Diversions may be nested. Certain parameters and registers are associated with the current diversion level.

Note: The top non-diversion level may be thought of as the 0th diversion level.

These parameters include the diversion trap and associated macro, `nospace` mode, the internally-saved marked place (see `mk` and `rt`), the current vertical place (`.d` register), the current high-water text baseline (`.h` register), and the current diversion name (`.z` register).

2.7.5 Traps

Three types of trap mechanisms are available: page traps, a diversion trap, and an input-line-count trap. Macro-invocation traps may be planted using **wh** at any page position including the top. This trap position may be changed using **ch**. Trap positions at or below the bottom of the page have no effect unless moved to within the page or rendered effective by an increase in page length. Two traps may be planted at the same position only by first planting them at different positions and then moving one of the traps; the first planted trap will conceal the second unless and until the first one is moved. If the first one is moved back, it again conceals the second trap. The macro associated with a page trap is automatically invoked when a line of text is output whose vertical size reaches or passes the trap position. Reaching the bottom of a page springs the top-of-page trap, if any, provided there is a next page. The distance to the next trap position is available in the **.t** register; if there are no traps between the current position and the bottom of the page, the distance returned in **.t** is the distance to the page bottom.

A macro-invocation trap effective in the current diversion may be planted using **dt**. The **.t** register works in a diversion; if there is no subsequent trap a large distance is returned. For a description of input-line-count traps, see **it** below.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes Explanation</i>
.de <i>xx yy</i> -	-	<i>.yy=..</i> -	Define or redefine the macro <i>xx</i> . The contents of the macro begin on the next input line. Input lines are copied in copy mode until the definition is terminated by a line beginning with <i>.yy</i> , whereupon the macro <i>yy</i> is called. In the absence of <i>yy</i> , the definition is terminated by a line beginning with <i>..</i> . A macro may contain de requests provided the terminating macros differ or the contained definition terminator is concealed. <i>..</i> can be concealed as <i>\\..</i> which will copy as <i>\\..</i> and be reread as <i>..</i> .
.am <i>xx yy</i> -	-	<i>.yy=..</i> -	Append to macro (append version of de).
.ds <i>xx st</i> -	-	ignored -	Define a string <i>xx</i> containing <i>st</i> . Any initial double-quote in <i>st</i> is stripped off to permit initial blanks.

.as <i>xx st</i> -	ignored -		Append <i>st</i> to string <i>xx</i> (append version of ds).
.rm <i>xx</i> -	ignored -		Remove request, macro, or string. The name <i>xx</i> is removed from the name list and any related storage space is freed. Subsequent references will have no effect.
.rn <i>xx yy</i> -	ignored -		Rename request, macro, or string <i>xx</i> to <i>yy</i> . If <i>yy</i> exists, it is first removed.
.di <i>xx</i> -	end	D	Divert output to macro <i>xx</i> . Normal text processing occurs during diversion except that page offsetting is not done. The diversion ends when the request di or da is encountered without an argument; extraneous requests of this type should not appear when nested diversions are being used.
.da <i>xx</i> -	end	D	Divert, appending to <i>xx</i> (append version of di).
.wh <i>N xx</i> -	-	v	Install a trap to invoke <i>xx</i> at page position <i>N</i> ; a negative <i>N</i> will be interpreted with respect to the page bottom. Any macro previously planted at <i>N</i> is replaced by <i>xx</i> . A zero <i>N</i> refers to the <i>top</i> of a page. In the absence of <i>xx</i> , the first trap found at <i>N</i> is removed.
.ch <i>xx N</i> -	-	v	Change the trap position for macro <i>xx</i> to be <i>N</i> . In the absence of <i>N</i> , the trap, if any, is removed.
.dt <i>N xx</i> -	off	D,v	Install a diversion trap at position <i>N</i> in the <i>current</i> diversion to invoke macro <i>xx</i> . Another dt will redefine the diversion trap. If no arguments are given, the diversion trap is removed.
.it <i>N xx</i> -	off	E	Set an input-line-count trap to invoke the macro <i>xx</i> after <i>N</i> lines of <i>text</i> input have been read (control or request lines don't count). The text may be in-line text or text interpolated by inline or trap-invoked macros.

.em *xx* none none - The macro *xx* will be invoked when all input has ended. The effect is the same as if the contents of *xx* had been at the end of the last file processed.

2.8 NUMBER REGISTERS

A variety of parameters are available as pre-defined, named number registers. In addition, the you may define your own named registers. Register names are one or two characters long and *do not* conflict with request, macro, or string names. Except for certain pre-defined, read-only registers, a number register can be read, written, automatically incremented or decremented, and interpolated into the input in a variety of formats. One common use of user-defined registers is to automatically number sections, paragraphs, lines, etc. A number register may be used any time numerical input is expected or desired and may be used in numerical expressions.

Number registers are created and modified using **nr**, which specifies the name, numerical value, and the auto-increment size. Registers are also modified, if accessed with an auto-incrementing sequence. If the registers *x* and *xx* both contain *N* and have the auto-increment size *M*, the following access sequences have the effect shown:

Sequence	Effect on Register	Value Interpolated
\nx	none	<i>N</i>
\n(xx	none	<i>N</i>
\n+x	<i>x</i> incremented by <i>M</i>	<i>N+M</i>
\n-x	<i>x</i> decremented by <i>M</i>	<i>N-M</i>
\n+(xx	<i>xx</i> incremented by <i>M</i>	<i>N+M</i>
\n-(xx	<i>xx</i> decremented by <i>M</i>	<i>N-M</i>

When interpolated, a number register is converted to decimal (default), decimal with leading zeros, lowercase Roman, uppercase Roman, lowercase sequential alphabetic, or uppercase sequential alphabetic according to the format specified by **af**.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
---------------------	----------------------	-----------------------	--------------	--------------------

.nr <i>R ±N M</i>		-	u	The number register <i>R</i> is assigned the value $\pm N$ with respect to the previous value, if any. The increment for auto-incrementing is set to <i>M</i> .
--------------------------	--	---	----------	---

.af <i>R c</i>	arabic	-	-	Assign format <i>c</i> to register <i>R</i> . The available formats are:
-----------------------	--------	---	---	---

Format	Numbering Sequence
1	0,1,2,3,4,5,...
001	000,001,002,003,004,005,...
i	0,i,ii,iii,iv,v,...
I	0,I,II,III,IV,V,...
a	0,a,b,c,....,z,aa,ab,....,zz,aaa,...
A	0,A,B,C,....,Z,AA,AB,....,ZZ,AAA,...

An arabic format having N digits specifies a field width of N digits. The read-only registers and the *width* function are always arabic.

.rr R - ignored - Remove register R . If many registers are being created dynamically, it may become necessary to remove unused registers to recapture storage space for new registers.

2.9 TABS, LEADERS, AND FIELDS

2.9.1 Tabs and Leaders

The ASCII horizontal tab character and the ASCII SOH (hereafter known as the “leader” character) can both be used to generate either horizontal motion or a string of repeated characters. The length of the generated entity is governed by internal tab stops specified with the **.ta** request. The default difference is that tabs generate motion and leaders generate a string of periods; **tc** and **lc** offer the choice of repeated character or motion. There are three types of internal tab stops: *left* adjusting, *right* adjusting, and *centering*. In the following table: D is the distance from the current position on the *input* line (where a tab or leader was found) to the next tab stop; *next-string* consists of the input characters following the tab (or leader) up to the next tab (or leader) or end of line; and W is the width of *next-string*.

Tab type	Length of motion or repeated characters	Location of <i>next-string</i>
Left	D	Following D
Right	$D-W$	Right adjusted within D
Centered	$D-W/2$	Centered on right end of D

The length of generated motion is allowed to be negative, but that of a repeated character string cannot be. Repeated character strings contain an integer number of characters, and any residual distance is prepended as motion. Tabs or leaders found after the last tab stop are ignored, but may be used as *next-string* terminators.

Tabs and leaders are not interpreted in copy mode. `\t` and `\a` always generate a non-interpreted tab and leader respectively, and are equivalent to actual tabs and leaders in copy mode.

2.9.2 Fields

A field is contained between a pair of field delimiter characters, and consists of sub-strings separated by padding indicator characters. The field length is the distance on the input line from the position where the field begins to the next tab stop. The difference between the total length of all the sub-strings and the field length is incorporated as horizontal padding space that is divided among the indicated padding places. The incorporated padding is allowed to be negative. For example, if the field delimiter is `#` and the padding indicator is `^`, `#^xxx^right#` specifies a right-adjusted string with the string `xxx` centered in the remaining space.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.ta Nt ...</code>	0.8; 0.5in	none	E,m	Set tab stops and types. <code>t=R</code> , right adjusting; <code>t=C</code> , centering; <code>t</code> absent, left adjusting. troff tab stops are preset every 0.5in.; nroff every 0.8in. The stop values are separated by spaces, and a value preceded by <code>+</code> is treated as an increment to the previous stop value.
<code>.tc c</code>	none	none	E	The tab repetition character becomes <code>c</code> , or is removed specifying motion.
<code>.lc c</code>	.	none	E	The leader repetition character becomes <code>c</code> , or is removed specifying motion.
<code>.fc a b</code>	off	off	-	The field delimiter is set to <code>a</code> ; the padding indicator is set to the <code>space</code> character or to <code>b</code> , if given. In the absence of arguments the field mechanism is turned off.

2.10 CONVENTIONS AND TRANSLATIONS

2.10.1 Input Character Translations

Normal character input has already been covered, as has input treatment of the ASCII control characters horizontal tab SOH and backspace. The newline delimits input lines. In addition, STX, ETX, ENQ, ACK, and BEL may be used as delimiters or translated into a graphic with `tr`. All other input characters are ignored.

The escape character `\` introduces escape sequences; it causes the following character to mean something else. There is a complete list of such sequences at the end of this chapter. The escape character `\` should not be confused with the ASCII control character ESC of the same name. The escape character itself can be input with the sequence `\\`. It can be changed with `ec`, and all that has been said about the default `\` becomes true for the new escape character. The request `\e` prints the current escape character. The escape mechanism may be turned off with `eo` and restored with `ec`.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.ec c</code>	<code>\</code>	<code>\</code>	-	Set escape character to <code>\</code> , or to <code>c</code> , if given.
<code>.eo</code>	on	-	-	Turn escape mechanism off.

2.10.2 Ligatures

Five ligatures are normally available: `fi`, `fl`, `ff`, `ffi`, and `ffl`. They may be input (even in `nroff`) by `\(fi`, `\(fl`, `\(ff`, `\(Fi`, and `\(Fl` respectively. The ligature mode is normally on in `troff`, and automatically invokes ligatures as needed.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.lg N</code>	off; on	on	-	Ligature mode is turned on if <code>N</code> is absent or non-zero, and turned off if <code>N=0</code> . If <code>N=2</code> , only the two-character ligatures are automatically invoked. Ligature mode is inhibited for request, macro, string, register, or file names, and in copy mode. No effect in <code>nroff</code> .

2.10.3 Backspacing, Underlining, Overstriking

Unless in copy mode, the ASCII backspace character is replaced by a backward horizontal motion having the width of the space character.

`Nroff` automatically underlines characters in the underline font, specifiable with `uf`, on font position 2 (normally Italic). In addition to `ft` and `\fF`, the underline font may be selected by `ul` and `cu`. Underlining is restricted to an output-device-dependent subset of characters.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.ul <i>N</i>	off	<i>N</i> =1	E	Underline in nroff (italicize in troff) the next <i>N</i> input text lines. Actually, switch to <i>underline</i> font, saving the current font for later restoration; other font changes within the span of a ul will take effect, but the restoration will undo the last change. Output generated by tl is affected by the font change, but does <i>not</i> decrement <i>N</i> . If <i>N</i> > 1, there is the risk that a trap interpolated macro may provide text lines within the span; environment switching can prevent this.
.cu <i>N</i>	off	<i>N</i> =1	E	A variant of ul that causes <i>every</i> character to be underlined in nroff . Identical to ul in troff .
.uf <i>F</i>	Italic	Italic	-	Underline font set to <i>F</i> . In nroff , <i>F</i> may <i>not</i> be on position 1 (initially Roman).

2.10.4 Request Characters

The default request character . (dot) and no-break request character ´ (acute accent) may be changed. Such a change, if undertaken, must be compatible with the design of any macros used during the span of the change, and particularly of any trap-invoked macros.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.cc <i>c</i>	.	.	E	The basic request character is set to <i>c</i> , or reset to ".".
.c2 <i>c</i>	´	´	E	The <i>nobreak</i> request character is set to <i>c</i> , or reset to "´".

2.10.5 Output Translation

One character can be made to stand-in for another character using **tr**. All text processing takes place with the input (stand-in) character which appears to have the width of the final (output) character. The graphic translation occurs at the moment of output (including diversion).

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.tr abcd...</code>	none	-	O	Translate <i>a</i> into <i>b</i> , <i>c</i> into <i>d</i> , etc. If an odd number of characters is given, the last one will be mapped into the space character. To be consistent, a particular translation must stay in effect from <i>input</i> to <i>output</i> time.

2.10.6 Transparent Throughput

An input line beginning with a `\!` is read in copy mode and transparently output (without the initial `\!`); `troff` is otherwise unaware of the line's presence. This mechanism may be used to pass control information to a post-processor or to imbed control lines in a macro created by a diversion.

2.10.7 Comments and Concealed Newlines

You may split a long input line that must stay one line (e. g. a string definition, or unfilled text) into many physical lines by ending all but the last one with the escape `\`. The sequence `\(newline)` is *always* ignored—except in a comment. Comments may be imbedded at the *end* of any line by prefacing them with `\`. The newline at the end of a comment cannot be concealed. A line beginning with `\` will appear as a blank line and behave like `.sp 1`; a comment can be on a line by itself by beginning the line with `.\`.

2.11 LOCAL MOTIONS

2.11.1 Local Motions

The functions `\v'N'` and `\h'N'` can be used for *local* vertical and horizontal motion respectively. The distance *N* may be negative; the *positive* directions are *rightward* and *downward*. A *local* motion is one contained *within* a line. To avoid unexpected vertical dislocations, be sure that the *net* vertical local motion within a word in filled text and otherwise within a line is zero. The escape sequences providing local motion are summarized in the following table.

Vertical Local Motion	Effect in		Horizontal Local Motion	Effect in	
	<i>troff</i>	<i>nroff</i>		<i>troff</i>	<i>nroff</i>
<code>\v'N'</code>	Move distance N		<code>\h'N'</code>	Move distance N	
<code>\u</code>	0.5 em up	0.5 line up	<code>\(space)</code>	Unpaddable space-size space	
<code>\d</code>	0.5 em down	0.5 line down	<code>\0</code>	Digit-size space	
<code>\r</code>	1 em up	1 line up	<code>\ </code>	1/6 em space	ignored
			<code>\^</code>	1/12 em space	ignored

As an example, E^2 can be generated by the following sequence.

$E\backslash s-2\backslash v'-0.4m'2\backslash v'0.4m'\backslash s+2$

It should be noted in this example that the 0.4 em vertical motions are at the smaller size.

2.11.2 The Width Function

The width function $\backslash w'$ *string* generates the numerical width of *string* (in basic units). Size and font changes may be safely imbedded in *string*, without affecting the current environment. For example, $.ti-\backslash w'1.'u$ could be used to temporarily indent leftward a distance equal to the size of the string "1."

The width function also sets three number registers. The registers *st* and *sb* are set respectively to the highest and lowest extent of *string* relative to the baseline; then, for example, the total height of the string is $\backslash n(stu-\backslash n(sbu$. In *troff*, the number register *ct* is set to a value between 0 and 3: 0 means that all of the characters in *string* were short lower-case characters without descenders (like *e*); 1 means that at least one character has a descender (like *y*); 2 means that at least one character is tall (like *H*); and 3 means that both tall characters and characters with descenders are present.

2.11.3 The Horizontal Place Marker

The escape sequence $\backslash kx$ will cause the current horizontal position in the input line to be stored in register *x*. As an example, the construction $\backslash kxword\backslash h'\backslash nxu+2u'word$ will embolden *word* by backing up to almost its beginning and overprinting it, resulting in *word*.

2.12 OVERSTRIKES, BRACKETS, AND LINES

2.12.1 Overstriking

The overstrike function allows centered overstriking of up to nine characters. It is invoked as $\backslash o'$ *string*'. The characters in *string* overprinted with centers aligned; the total width is that of the widest character. *String* cannot contain local vertical motion. As examples, $\backslash o'e''$ produces \acute{e} , and $\backslash o'\backslash (mo\backslash (sl'$ produces \notin .

2.12.2 Zero-Width Characters


The function $\backslash zc$ will output *c* without spacing over it, and can be used to produce left-aligned overstruck combinations. As examples, $\backslash z\backslash (ci\backslash (pl$ will produce \oplus , and $\backslash (br\backslash z\backslash (rn\backslash (ul\backslash (br$ will produce the smallest possible constructed box \square .

2.12.3 Large Brackets

The Special Mathematical Font contains a number of bracket construction pieces ($($ $)$ $\{$ $\}$ $|$ $]$ $[$) that can be combined into various bracket styles. The function $\backslash b'$ *string*' may be used to pile up vertically the characters in *string* (the first character on top and the last at the bottom); the characters are vertically separated by 1 em and the

total pile is centered 1/2 em above the current baseline (line in **nroff**).
For example,

```
\b' \lc \lf ' E \ b' \rc \rf ' \x' -0.5m' \x' 0.5m'
```

produces .

2.12.4 Line Drawing

The function `\l' Nc'` will draw a string of repeated *c*'s towards the right for a distance *N*. (`\l` is `\(` lower-case L, not number 1). If *c* looks like a continuation of an expression for *N*, it may be insulated from *N* with a `\&`. If *c* is not specified, the `_` (baseline rule) is used (underline character in **nroff**). If *N* is negative, a backward horizontal motion of size *N* is made before drawing the string. Any space resulting from *N*/(size of *c*) having a remainder is put at the beginning (left end) of the string. In the case of characters that are designed to be connected such as baseline-rule `_`, underrule `_`, and root-en `ˉ`, the remaining space is covered by overlapping. If *N* is *less* than the width of *c*, a single *c* is centered on a distance *N*. As an example, a macro to underscore a string can be written

```
.de us
\\$1\l' 0\ul'
..
```

or one to draw a box around a string

```
.de bx
\\(br\ \\$1\ \\(br\l' 0\ (rn' \l' 0\ul'
..
```

such that

```
.ul "underlined words"
```

and

```
.bx "words in a box"
```

yield underlined words and words in a box.

The function `\L' Nc'` will draw a vertical line consisting of the (optional) character *c* stacked vertically apart 1 em (1 line in **nroff**), with the first two characters overlapped, if necessary, to form a continuous line. The default character is the *box rule* `|` (`\(br`); the other suitable character is the *bold vertical* `|` (`\(bv`). The line is begun without any initial motion relative to the current baseline. A positive *N* specifies a line drawn downwards and a negative *N* specifies a line drawn upwards. After the line is drawn *no* compensating motions are made; the current baseline is at the *end* of the line.

The horizontal and vertical line drawing functions may be used in combination to produce large boxes. The zero-width box-rule and the em wide underrule were designed to form corners when using 1-em vertical spacings. For example the macro

```
.de eb
\'compensate for next automatic baseline spacing
.sp -1
\'avoid possibly overflowing word buffer
.nf
\'do box
\'h'- .5n' \L' \| \nau-1' \l' \| n(.lu+1n) (\ul' \L' - \| \nau+1' \l' | 0u- .5n) (\ul' .tr |
.fi
..
```

will draw a box around some text whose beginning vertical place was saved in number register *a* (e.g. using `.mk|a`) as was done for this paragraph.

2.13 HYPHENATION

The automatic hyphenation feature may be switched off and on as needed. When switched on with `hy`, several variants may be set. A hyphenation indicator character may be imbedded in a word to specify desired hyphenation points, or may be prepended to suppress hyphenation. In addition, you may specify a small exception word list.

Only words that consist of a central alphabetic string surrounded by (usually null) non-alphabetic strings are considered candidates for automatic hyphenation. Words that were input containing hyphens (minus signs), em-dashes (`\(em)`), or hyphenation indicator characters, such as mother-in-law, are *always* subject to splitting after those characters, whether or not automatic hyphenation is on.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.nh</code>	hyphenate -		E	Automatic hyphenation is turned off.
<code>.hyN</code>	on, N=1	on, N=1	E	Automatic hyphenation is turned on for $N \geq 1$, or off for $N=0$. If $N=2$, <i>last</i> lines (ones that will cause a trap) are not hyphenated. For $N=4$ and 8 , the last and first two characters respectively of a word are not split off. These values are additive; i. e. $N=14$ will invoke all three restrictions.
<code>.hc c</code>	<code>\%</code>	<code>\%</code>	E	Hyphenation indicator character is set to <i>c</i> or to the default <code>\%</code> . The indicator does not appear in

the output.

.hw word1 ... ignored - Specify hyphenation points in words with imbedded minus signs. Versions of a word with terminal *s* are implied; i. e. *dig-it* implies *dig-its*. This list is examined initially *and* after each suffix stripping. The space available is small, about 128 characters.

2.14 THREE-PART TITLES

The titling function **tl** provides for automatic placement of three fields at the left, center, and right of a line. The title-length is specified with **lt**, which may be used anywhere, and is independent of the normal text collecting process. A common use of **lt** is in header and footer macros.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.tl 'left'center'right'		-	-	The strings <i>left</i> , <i>center</i> , and <i>right</i> are respectively left-adjusted, centered, and right-adjusted in the current title-length. Any of the strings may be empty, and overlapping is permitted. If the page-number character (initially %) is found within any of the fields, it is replaced by the current page number having the format assigned to register % . Any character may be used as the string delimiter.
.pc <i>c</i>	%	off	-	The page number character is set to <i>c</i> , or removed. The page-number register remains % .
.lt $\pm N$	6.5 in	previous	E,m	Length of title set to $\pm N$. The line-length and the title-length are independent. Indents do not apply to titles; page-offsets do.

2.15 OUTPUT LINE NUMBERING

Automatic numbering of output lines may be requested with **nm**. When in effect, a three-digit, arabic number plus a digit-space is prepended to output text lines. The text lines are thus offset by four digit-spaces, although they retain their line length. A reduction in line length may be desired to keep the right margin aligned with an earlier margin. Blank

lines, other vertical spaces, and lines generated by **tl** are not numbered. Numbering can be temporarily suspended with **nn**, or with an **.nm** followed by a later **.nm +0**. In addition, a line number indent *I*, and the number-text separation *S* may be specified in digit-spaces. Further, it can be specified that only those line numbers that are multiples of some number *M* are to be printed (the others will appear as blank number fields).

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.nm $\pm N M S I$		off	E	Line number mode. If $\pm N$ is given, line numbering is turned on, and the next output line numbered is numbered $\pm N$. Default values are $M=1$, $S=1$, and $I=0$. Parameters corresponding to missing arguments are unaffected; a non-numeric argument is considered missing. In the absence of all arguments, numbering is turned off; the next line number is preserved for possible further use in number register ln .
.nn <i>N</i>	-	$N=1$	E	The next <i>N</i> text output lines are not numbered.

2.16 CONDITIONAL ACCEPTANCE OF INPUT

In the following discussion, *c* is a one-character, built-in *condition* name, **!** signifies *not*, *N* is a numerical expression, *string1* and *string2* are strings delimited by any non-blank, non-numeric character *not* in the strings, and *anything* represents what is conditionally accepted.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.if <i>c anything</i>	-	-	-	If condition <i>c</i> true, accept <i>anything</i> as input; in multi-line case use $\backslash\{anything\}$.
.if ! <i>c anything</i>	-	-	-	If condition <i>c</i> false, accept <i>anything</i> .
.if <i>N anything</i>	-	-	u	If expression $N > 0$, accept <i>anything</i> .
.if ! <i>N anything</i>	-	-	u	If expression $N \leq 0$, accept <i>anything</i> .
.if ' <i>string1</i> ' <i>string2</i> ' <i>anything</i>	-	-	-	If <i>string1</i> identical to <i>string2</i> , accept <i>anything</i> .

- .if** *!' string1' string2' anything* - If *string1* not identical to *string2*, accept *anything*.
- .ie** *c anything* - **u** If portion of if-else; all above forms (like **if**).
- .el** *anything* - - Else portion of if-else.

2.16.1 Built-In Conditions

There are several conditions of which **troff** is always aware. These built-in condition names are:

Condition	True If
o	Current page number is odd
e	Current page number is even
t	Formatter is troff
n	Formatter is nroff

If the condition *c* is *true*, or if the number *N* is greater than zero, or if the strings compare identically (including motions and character size and font), *anything* is accepted as input. If a **!** precedes the condition, number, or string comparison, the sense of the acceptance is reversed.

Any spaces between the condition and the beginning of *anything* are skipped over. The *anything* can be either a single input line (text, macro, or whatever) or a number of input lines. In the multi-line case, the first line must begin with a left delimiter **{** and the last line must end with a right delimiter **}**.

The request **ie** (if-else) is identical to **if** except that the acceptance state is remembered. A subsequent and matching **el** (else) request then uses the reverse sense of that state. **ie** - **el** pairs may be nested.

Some examples are:

```
.if e .tl 'Even Page %'
```

which outputs a title if the page number is even; and

```
.ie \n%>1 \{\  
'sp 0.5i  
.tl 'Page %'  
'sp 1.2i \}  
.el .sp 2.5i
```

which treats page 1 differently from other pages.

2.17 ENVIRONMENTS

A number of the parameters that control the text processing are gathered together into an environment, which can be switched by the user. The environment parameters are those associated with requests noting **E** in their *Notes* column; in addition, partially collected lines and words are in the environment. Everything else is global; examples are page-oriented

parameters, diversion-oriented parameters, number registers, and macro and string definitions. All environments are initialized with default parameter values.

<i>Request</i>	<i>Initial</i>	<i>If No</i>		
<i>Form</i>	<i>Value</i>	<i>Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.ev N</code>	$N=0$	previous	-	Environment switched to environment $0 \leq N \leq 2$. Switching is done in push-down fashion so that restoring a previous environment must be done with <code>.ev</code> rather than specific reference.

2.18 INSERTIONS FROM STANDARD INPUT

The input can be temporarily switched to the system standard input with the `rd` request, which will switch back when two newlines in a row are read (the extra blank line is not used). This mechanism is useful for things like forms and form letters.

<i>Request</i>	<i>Initial</i>	<i>If No</i>		
<i>Form</i>	<i>Value</i>	<i>Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.rd p</code>	-	$p=BEL$	-	Read insertion from the standard input until two newlines in a row are found. If the standard input is the user's keyboard, p (or a BEL) is written onto the user's terminal. <code>rd</code> behaves like a macro, and arguments may be placed after p .
<code>.ex</code>	-	-	-	Exit from <code>nroff/troff</code> . Text processing is terminated exactly as if all input had ended.

2.18.1 Prompts

If insertions are to be taken from the terminal keyboard while output is being printed on the terminal, the command line option `-q` will turn off the echoing of keyboard input and prompt only with BEL. The regular input and insertion input cannot be simultaneously read from the standard input.

As an example, multiple copies of a form letter may be prepared by entering the insertions for all the copies in one file to be used as the standard input, and causing the file containing the letter to reinvoke itself using `nx`. The process would ultimately be ended by an `ex` in the insertion file.

2.19 INPUT/OUTPUT FILE SWITCHING

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.so <i>filename</i>	-	-	-	Switch source file. The top input (file reading) level is switched to <i>filename</i> . The effect of a so encountered in a macro is not felt until the input level returns to the file level. When the new file ends, input is again taken from the original file. so lines may be nested.
.nx <i>filename</i>	-	end-of-file	-	Next file is <i>filename</i> . The current file is considered ended, and the input is immediately switched to <i>filename</i> .
.pi <i>program</i>	-	-	-	Pipe output to <i>program</i> (nroff only). This request must occur before any printing occurs. No arguments are transmitted to <i>program</i> .

2.20 MISCELLANEOUS REQUESTS

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.mc <i>c N</i>	-	off	E,m	Specifies that a margin character <i>c</i> appear a distance <i>N</i> to the right of the right margin after each non-empty text line (except those produced by tl). If the output line is too long (as can happen in nofill mode) the character will be appended to the line. If <i>N</i> is not given, the previous <i>N</i> is used; the initial <i>N</i> is 0.2 inches in nroff and 1 em in troff . The margin character used with this paragraph was a 14-point box-rule.
.tm <i>string</i>	-	newline	-	After skipping initial blanks, <i>string</i> (rest of the line) is read in copy mode and written on the standard message output.
.ig <i>yy</i>	-	<i>.yy=.</i>	-	Ignore input lines. ig behaves exactly like de except that the

				input is discarded. The input is read in <i>copy mode</i> , and any auto-incremented registers will be affected.	
.pm	<i>t</i>	-	all	-	Print macros. The names and sizes of all of the defined macros and strings are printed on the terminal; if <i>t</i> is given, only the total of the sizes is printed. The sizes is given in blocks of 128 characters.
.fl		-		B	Flush output buffer. Used in interactive debugging to force output.

2.21 OUTPUT AND ERROR MESSAGES

The output from **tm**, **pm**, and the prompt from **rd**, as well as various error messages are written to message (or error) output. By default, both standard and error output appear on the terminal (the transcript pad). The Bourne shell allows error output to be independantly redirected. The C shell does not.

Various error conditions may occur during the operation of **nroff** and **troff**. Errors that have only local impact do not cause processing to terminate. Two examples are word overflow, caused by a word that is too large to fit into the word buffer (in fill mode), and line overflow, caused by an output line that grew too large to fit in the line buffer; in both cases, a message is printed, the offending excess is discarded, and the affected word or line is marked at the point of truncation with a * in **nroff** and a ← in **troff**. The aim is to continue processing, if possible, since any output produced may be useful for debugging. If a serious error occurs, processing terminates, and an appropriate message is printed. Examples are the inability to create, read, or write files, and the exceeding of certain internal limits that make future output unlikely to be useful.

2.22 FONT STYLE EXAMPLES

The following fonts are printed in 12-point, with a vertical spacing of 14-point, and with non-alphanumeric characters separated by a quarter-em space.

Times Roman

abcdefghijklmnopqrstvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
!\$%&'()*+,-./:;=?[]|
• - - - ¼ fi fl ff ffi ffl ° † ' ©

Times Italic

abcdefghijklmnopqrstvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
!\$%&'()*+,-./:;=?[]|
• - - - fi fl ff ffi ffl ° † ' ©

Times Bold

abcdefghijklmnopqrstvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
!\$%&'()*+,-./:;=?[]|
• - - - fi fl ff ffi ffl ° † ' ©

Special Mathematical Font

" '\ ^ _ ` ~ / < > { } # @ + - = *
α β γ δ ε ζ η θ ι κ λ μ
ν ξ ο π ρ σ τ υ φ χ ψ ω
Γ Δ Θ Λ Ξ Π Σ Υ Φ Ψ Ω
√ ∓ ≥ ≤ ≡ ~ ≈ ≠ → ← ↑ ↓ × ÷ ± ∪ ∩ ⊃ ⊆ ∞ ∂
§ ∇ ∇ ∫ ≈ ∅ ∈ ‡ ⇒ ⇐ ⊙ | ○ √ ∪ ∩ ∪ ∩ ∪ ∩

2.23 INPUT CHARACTER NAMES

2.23.1 Special Characters on Standard Fonts

This table indicates various escape sequences that you can use to obtain characters that aren't on most terminal keyboards.

Note: The availability of output characters varies from device to device. If the character is not available, it prints as a 1-em space, as can be seen in the following examples of output from an IMAGEN CX laser printer.

<i>Character (glyph)</i>	<i>Input Name</i>	<i>Character Name</i>
'	'	close quote
`	`	open quote
—	\\(em	3/4 Em dash
-	-	hyphen
-	\\(hy	hyphen
-	/-	current font minus
•	\\(bu	bullet
□	\\(sq	square
—	\\(ru	rule
¼	\\(14	1/4
	\\(12	1/2
	\\(34	3/4
fi	\\(fi	fi
fl	\\(fl	fl
ff	\\(ff	ff
ffi	\\(Fi	ffi
ffl	\\(Fl	ffl
°	\\(de	degree
†	\\(dg	dagger
'	\\(fm	foot mark
	\\(ct	cent sign
	\\(rg	registered
©	\\(co	copyright

2.23.2 Characters on the Special Font

Many non-ASCII characters and the ASCII characters ' , ` , _ , + , - , = , and * exist on the special font. The ASCII characters @ , # , " , ' , ` , < , > , \ , { , } , ~ , ^ , and _ exist *only* on the special font and are printed as a 1-em space if that font is not mounted. The following characters exist only on the special font except for the upper case Greek letter names. These are mapped into upper case English letters in whatever font is mounted on font position one (default Roman). The special math plus, minus, and equals are provided to insulate the appearance of equations from the choice of standard fonts.

<i>Character (glyph)</i>	<i>Input Name</i>	<i>Character Name</i>
+	<code>\(pl</code>	math plus
-	<code>\(mi</code>	math minus
=	<code>\(eq</code>	math equals
*	<code>\(**</code>	math star
§	<code>\(sc</code>	section
'	<code>\(aa</code>	acute accent
`	<code>\(ga</code>	grave accent
—	<code>\(ul</code>	underrule
/	<code>\(sl</code>	slash (matching backslash)
α	<code>\(*a</code>	alpha
β	<code>\(*b</code>	beta
γ	<code>\(*g</code>	gamma
δ	<code>\(*d</code>	delta
ε	<code>\(*e</code>	epsilon
ζ	<code>\(*z</code>	zeta
η	<code>\(*y</code>	eta
θ	<code>\(*h</code>	theta
ι	<code>\(*i</code>	iota
κ	<code>\(*k</code>	kappa
λ	<code>\(*l</code>	lambda
μ	<code>\(*m</code>	mu
ν	<code>\(*n</code>	nu
ξ	<code>\(*c</code>	xi
ο	<code>\(*o</code>	omicron
π	<code>\(*p</code>	pi
ρ	<code>\(*r</code>	rho
σ	<code>\(*s</code>	sigma
φ	<code>\(ts</code>	terminal sigma
τ	<code>\(*t</code>	tau
υ	<code>\(*u</code>	upsilon
φ	<code>\(*f</code>	phi
χ	<code>\(*x</code>	chi
ψ	<code>\(*q</code>	psi
ω	<code>\(*w</code>	omega
A	<code>\(*A</code>	Alpha
B	<code>\(*B</code>	Beta
Γ	<code>\(*G</code>	Gamma
Δ	<code>\(*D</code>	Delta
E	<code>\(*E</code>	Epsilon
Z	<code>\(*Z</code>	Zeta
H	<code>\(*Y</code>	Eta
Θ	<code>\(*H</code>	Theta
I	<code>\(*I</code>	Iota
K	<code>\(*K</code>	Kappa
Λ	<code>\(*L</code>	Lambda
M	<code>\(*M</code>	Mu
N	<code>\(*N</code>	Nu
Ξ	<code>\(*C</code>	Xi
O	<code>\(*O</code>	Omicron
Π	<code>\(*P</code>	Pi
P	<code>\(*R</code>	Rho

<i>Character (glyph)</i>	<i>Input Name</i>	<i>Character Name</i>
Σ	<code>\(*S</code>	Sigma
τ	<code>\(*T</code>	Tau
Υ	<code>\(*U</code>	Upsilon
Φ	<code>\(*F</code>	Phi
χ	<code>\(*X</code>	Chi
Ψ	<code>\(*Q</code>	Psi
Ω	<code>\(*W</code>	Omega
$\sqrt{\quad}$	<code>\(sr</code>	square root
$\sqrt[n]{\quad}$	<code>\(rn</code>	root en extender
$>=$	<code>\(>=</code>	$>=$
$<=$	<code>\(<=</code>	$<=$
\equiv	<code>\(==</code>	identically equal
\approx	<code>\(^ =</code>	approx =
\sim	<code>\(ap</code>	approximates
\neq	<code>\(!=</code>	not equal
\rightarrow	<code>\(-></code>	right arrow
\leftarrow	<code>\(<-</code>	left arrow
\uparrow	<code>\(ua</code>	up arrow
\downarrow	<code>\(da</code>	down arrow
\times	<code>\(mu</code>	multiply
\div	<code>\(di</code>	divide
\pm	<code>\(+-</code>	plus-minus
\cup	<code>\(cu</code>	cup (union)
\cap	<code>\(ca</code>	cap (intersection)
\subset	<code>\(sb</code>	subset of
\supset	<code>\(sp</code>	superset of
\subsetneq	<code>\(ib</code>	improper subset
\supsetneq	<code>\(ip</code>	improper superset
∞	<code>\(if</code>	infinity
∂	<code>\(pd</code>	partial derivative
∇	<code>\(gr</code>	gradient
\neg	<code>\(no</code>	not
\int	<code>\(is</code>	integral sign
\propto	<code>\(pt</code>	proportional to
\emptyset	<code>\(es</code>	empty set
\in	<code>\(mo</code>	member of
$\rule{0pt}{1ex}$	<code>\(br</code>	box vertical rule
\ddagger	<code>\(dd</code>	double dagger
\rightrightarrows	<code>\(rh</code>	right hand
\leftrightharpoons	<code>\(lh</code>	left hand
\circ	<code>\(bs</code>	bell
---	<code>\(or</code>	or
\bigcirc	<code>\(ci</code>	circle

Character (glyph)	Input Name	Character Name
{	\{lt	left top of big curly bracket
{	\{lb	left bottom
}	\}rt	right top
}	\}rb	right bot
{	\{lk	left center of big curly bracket
}	\}rk	right center of big curly bracket
	\ bv	bold vertical
	\ lf	left floor (left bottom of big square bracket)
	\ rf	right floor (right bottom)
	\ lc	left ceiling (left top)
	\ rc	right ceiling (right top)

2.24 SUMMARY OF REQUESTS

Values separated by ";" are for **nroff** and **troff** respectively. The following designation are used in the *Notes* column.

- B** Request normally causes a break.
- D** Mode or relevant parameters associated with current diversion level.
- E** Relevant parameters are a part of the current environment.
- O** Must stay in effect until logical output.
- P** Mode must be still or again in effect at the time of physical output.

The characters **v,p,m,u** are default scale indicators; if not specified, scale indicators are *ignored*.

2.24.1 Font and Character Size Control

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.ps $\pm N$	10 point	previous	E	Point size; also $\backslash s\pm N$.
.ss N	12/36 em	ignored	E	Space-character size set to $N/36$ em.
.cs FNM	off	-	P	Constant character space (width) mode (font F).
.bd $F N$	off	-	P	Embolden font F by $N-1$ units.
.bd $S F N$ off		-	P	Embolden Special Font when current font is F .
.ft F	Roman	previous	E	Change to font $F = x, xx,$ or 1-4. Also $\backslash fx, \backslash f(xx, \backslash fN$.
.fp $N F$	R,I,B,S	ignored	-	Font named F mounted on physical position $1 \leq N \leq 4$.

2.24.2 Page Control

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.pl ± <i>N</i>	11 in	11 in	v	Page length.
.bp ± <i>N</i>	<i>N</i> =1	-	B†,v	Eject current page; next page number <i>N</i> .
.pn ± <i>N</i>	<i>N</i> =1	ignored	-	Next page number <i>N</i> .
.po ± <i>N</i>	0; 26/27 in	previous	v	Page offset.
.ne <i>N</i>	-	<i>N</i> =1 <i>V</i>	D,v	Need <i>N</i> vertical space (<i>V</i> = vertical spacing).
.mk <i>R</i>	none	internal	D	Mark current vertical place in register <i>R</i> .
.rt ± <i>N</i>	none	internal	D,v	Return (<i>upward only</i>) to marked vertical place.

2.24.3 Text Filling, Adjusting, and Centering

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.br	-	-	B	Break.
.fi	fill	-	B,E	Fill output lines.
.nf	fill	-	B,E	No filling or adjusting of output lines.
.ad <i>c</i>	adj,both	adjust	E	Adjust output lines with mode <i>c</i> .
.na	adjust	-	E	No output line adjusting.
.ce <i>N</i>	off	<i>N</i> =1	B,E	Center following <i>N</i> input text lines.

2.24.4 Vertical Spacing

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.vs <i>N</i>	12pts	previous	E,p	Vertical base line spacing (<i>V</i>).
.ls <i>N</i>	<i>N</i> =1	previous	E	Output <i>N</i> -1 <i>V</i> s after each text output line.
.sp <i>N</i>	-	<i>N</i> =1 <i>V</i>	B,v	Space vertical distance <i>N</i> in <i>either direction</i> .
.sv <i>N</i>	-	<i>N</i> =1 <i>V</i>	v	Save vertical distance <i>N</i> .
.os	-	-	-	Output saved vertical distance.
.ns	space	-	D	Turn no-space mode on.
.rs	-	-	D	Restore spacing; turn no-space mode off.

2.24.5 Line Length and Indenting

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.ll $\pm N$	6.5 in	previous	E,m	Line length.
.in $\pm N$	$N=0$	previous	B,E,m	Indent.
.ti $\pm N$	-	ignored	B,E,m	Temporary indent.

2.24.6 Macros, Strings, Diversions, Traps

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.de $xx\ yy$	-	.yy=..	-	Define or redefine macro xx ; end at call of yy .
.am $xx\ yy$	-	.yy=..	-	Append to a macro.
.ds $xx\ st$	-	ignored	-	Define a string xx containing st .
.as $xx\ st$	-	ignored	-	Append st to string xx .
.rm xx	-	ignored	-	Remove request, macro, or string.
.rn $xx\ yy$	-	ignored	-	Rename request, macro, or string xx to yy .
.di xx	-	end	D	Divert output to macro xx .
.da xx	-	end	D	Divert and append to xx .
.wh $N\ xx$	-	-	v	Set location trap; negative is w.r.t. page bottom.
.ch $xx\ N$	-	-	v	Change trap location.
.dt $N\ xx$	-	off	D,v	Set a diversion trap.
.it $N\ xx$	-	off	E	Set an input-line count trap.
.em xx	none	none	-	End macro is xx .

2.24.7 Number Registers

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.nr $R\ \pm N\ M$	-	-	u	Define and set number register R ; auto-increment by M .
.af $R\ c$	arabic	-	-	Assign format to register R ($c=1, i, I, a, A$).
.rr R	-	-	-	Remove register R .

2.24.8 Tabs, Leaders, and Fields

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<i>.ta Nt ...</i>	0.8; 0.5in	none	E,m	Tab settings; <i>left</i> type, unless <i>t=R</i> (right), <i>C</i> (centered).
<i>.tc c</i>	none	none	E	Tab repetition character.
<i>.lc c</i>	.	none	E	Leader repetition character.
<i>.fc a b</i>	off	off	-	Set field delimiter <i>a</i> and pad character <i>b</i> .

2.24.9 I/O Conventions and Translations

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<i>.ec c</i>	\	\	-	Set escape character.
<i>.eo</i>	on	-	-	Turn off escape character mechanism.
<i>.lg N</i>	-; on	on	-	Ligature mode on if $N > 0$.
<i>.ul N</i>	off	$N=1$	E	Underline (italicize in troff) N input lines.
<i>.cu N</i>	off	$N=1$	E	Continuous underline in nroff ; like ul in troff .
<i>.uf F</i>	Italic	Italic	-	Underline font set to <i>F</i> (to be switched to by ul).
<i>.cc c</i>	.	.	E	Set control character to <i>c</i> .
<i>.c2 c</i>	,	,	E	Set nobreak control character to <i>c</i> .
<i>.tr abcd....</i>	none	-	O	Translate <i>a</i> to <i>b</i> , etc. on output.

2.24.10 Hyphenation

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<i>.nh</i>	hyphenate	-	E	No hyphenation.
<i>.hy N</i>	hyphenate	hyphenate	E	Hyphenate; $N =$ mode.
<i>.hc c</i>	\%	\%	E	Hyphenation indicator character <i>c</i> .
<i>.hw word1 ...</i>		ignored	-	Exception words.

2.24.11 Three Part Titles

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.tl 'l'c'r'</code>		-	-	Three-part title.
<code>.pc c</code>	%	off	-	Page number character.
<code>.lt ±N</code>	6.5 in	previous	E,m	Length of title.

2.24.12 Output Line Numbering

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.nm ±N M S I</code>		off	E	Number mode on or off, set parameters.
<code>.nn N</code>	-	N=1	E	Do not number next N lines.

2.24.13 Conditional Acceptance of Input

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.if c anything</code>		-	-	If condition <i>c</i> true, accept <i>anything</i> as input, for multi-line use <code>\{anything\}</code> .
<code>.if !c anything</code>		-	-	If condition <i>c</i> false, accept <i>anything</i> .
<code>.if N anything</code>		-	u	If expression $N > 0$, accept <i>anything</i> .
<code>.if !N anything</code>		-	u	If expression $N \leq 0$, accept <i>anything</i> .
<code>.if 'string1' string2' anything</code>		-	-	If <i>string1</i> identical to <i>string2</i> , accept <i>anything</i> .
<code>.if !'string1' string2' anything</code>		-	-	If <i>string1</i> not identical to <i>string2</i> , accept <i>anything</i> .
<code>.ie c anything</code>		-	u	If portion of if-else; all above forms (like <code>if</code>).
<code>.el anything</code>		-	-	Else portion of if-else.

2.24.14 Environment Switching

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.ev <i>N</i>	<i>N=0</i>	previous	-	Environment switched (<i>push down</i>).

2.24.15 Insertions from the Standard Input

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.rd <i>p</i>	-	<i>p=BEL</i>	-	Read insertion.
.ex	-	-	-	Exit from nroff/troff .

2.24.16 Input/Output File Switching

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.so <i>file</i>	-	-	-	Switch source file (<i>push down</i>).
.nx <i>file</i>	<i>eof</i>	-	-	Next file.
.pi <i>program</i>	-	-	-	Pipe output to <i>program</i> (nroff only).

2.24.17 Miscellaneous Requests

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.mc <i>c N</i>	-	<i>off</i>	<i>E,m</i>	Set margin character <i>c</i> and separation <i>N</i> .
.tm <i>string</i>	-	<i>newline</i>	-	Print <i>string</i> on terminal (UNIX standard message output).
.ig <i>yy</i>	-	<i>.yy=..</i>	-	Ignore till call of <i>yy</i> .
.pm <i>t</i>	-	<i>all</i>	-	Print macro names and sizes; if <i>t</i> present, print only total of sizes.
.fl	-	-	<i>B</i>	Flush output buffer.

2.25 SUMMARY OF ESCAPE SEQUENCES

<i>Escape Sequence</i>	<i>Meaning</i>
\\	\ (to prevent or delay the interpretation of \)
\e	Printable version of the <i>current</i> escape character.
\`	` (acute accent); equivalent to \(\aa
\~	~ (grave accent); equivalent to \(\ga
\-	- Minus sign in the <i>current</i> font
\.	Period (dot) (see <i>de</i>)
\(space)	Unpaddable space-size space character
\0	Digit width space
\^	1/6 em narrow space character (zero width in <i>nroff</i>)
\^	1/12 em half-narrow space character (zero width in <i>nroff</i>)
\.	Non-printing, zero width character
\!	Transparent line indicator
\"	Beginning of comment
\\$N	Interpolate argument $1 \leq N \leq 9$
\%	Default optional hyphenation character
\(xx	Character named <i>xx</i>
*x, *(xx	Interpolate string <i>x</i> or <i>xx</i>
\a	Non-interpreted leader character
\b'abc...'	Bracket building function
\c	Interrupt text processing
\d	Forward (down) 1/2 em vertical motion (1/2 line in <i>nroff</i>)
\fx, \f(xx, \fN	Change to font named <i>x</i> or <i>xx</i> , or position <i>N</i>
\h'N'	Local horizontal motion; move right <i>N</i> (<i>negative left</i>)
\kx	Mark horizontal <i>input</i> place in register <i>x</i>
\l'Nc'	Horizontal line drawing function (optionally with <i>c</i>)
\L'Nc'	Vertical line drawing function (optionally with <i>c</i>)
\nx, \n(xx	Interpolate number register <i>x</i> or <i>xx</i>
\o'abc...'	Overstrike characters <i>a</i> , <i>b</i> , <i>c</i> , ...
\p	Break and spread output line
\r	Reverse 1 em vertical motion (reverse line in <i>nroff</i>)
\sN, \s±N	Point-size change function
\t	Non-interpreted horizontal tab
\u	Reverse (up) 1/2 em vertical motion (1/2 line in <i>nroff</i>)
\v'N'	Local vertical motion; move down <i>N</i> (<i>negative up</i>)
\w'string'	Interpolate width of <i>string</i>
\x'N'	Extra line-space function (<i>negative before, positive after</i>)
\zc	Print <i>c</i> with zero width (without spacing)
\{	Begin conditional input
\}	End conditional input
\(newline)	Concealed (ignored) newline
\X	<i>X</i> , any character <i>not</i> listed above

Note: The escape sequences \\, \., \", \\$, *, \a, \n, \t, and \(\newline) are interpreted in *copy mode*.

2.26 SUMMARY OF PRE-DEFINED GENERAL NUMBER REGISTERS

<i>Register Name</i>	<i>Description</i>
%	
ct	Character type (set by <i>width</i> function).
dl	Width (maximum) of last completed diversion.
dn	Height (vertical size) of last completed diversion.
dw	Current day of the week (1-7).
dy	Current day of the month (1-31).
hp	Current horizontal place on <i>input</i> line.
ln	Output line number.
mo	Current month (1-12).
nl	Vertical position of last printed text baseline.
sb	Depth of string below base line (generated by <i>width</i> function).
st	Height of string above base line (generated by <i>width</i> function).
yr	Last two digits of current year.

2.27 SUMMARY OF PRE-DEFINED READ-ONLY NUMBER REGISTERS

<i>Register Name</i>	<i>Description</i>
.\$	Number of arguments available at the current macro level.
.A	Set to 1 in troff , if -a option used; always 1 in nroff .
.H	Available horizontal resolution in basic units.
.T	Set to 1 in nroff , if -T option used; always 0 in troff .
.V	Available vertical resolution in basic units.
.a	Post-line extra line-space most recently utilized B\x'N'.
.c	Number of <i>lines</i> read from current input file.
.d	Current vertical place in current diversion; equal to nl , if no diversion.
.f	Current font as physical quadrant (1-4):
.h	Text baseline high-water mark on current page or diversion.
.i	Current indent.
.l	Current line length.
.n	Length of text portion on previous output line.
.o	Current page offset.
.p	Current page length.
.s	Current point size.
.t	Distance to the next trap.
.u	Equal to 1 in fill mode and 0 in nofill mode.
.v	Current vertical line spacing.
.w	Width of previous character.
.x	Reserved version-dependent register.
.y	Reserved version-dependent register.
.z	Name of current diversion.

C

C

C

C

C

Chapter 3: The **-ms** Macro Package

3.1 INTRODUCTION

The **-ms** macros were developed at Bell Telephone Laboratories for the purpose of formatting a variety of papers and memoranda. The package was further enhanced at University of California at Berkeley. The *bsd4.2* version of DOMAIN/IX includes both the “old” and “new” versions of **-ms**. The *sys5* version of DOMAIN/IX includes only the “old” **-ms**.

The new **-ms** macros have been slightly revised and rearranged. As a result, they can be read by the computer in about half the time required by previous versions of **-ms**. This means that output will begin to appear sooner, especially when a small file is being processed. The old version of **-ms** is still available to DOMAIN/IX *bsd4.2* users. It resides in the file */usr/lib/tmac/tmac.os* and can be invoked by including the flag **-mos** on the **n/troff** command line.

In the new version, several bugs have been fixed, including a problem with the **.1C** macro, minor difficulties with boxed text, a break induced by **.EQ** before initialization, the failure to set tab stops in displays, and several bothersome errors in the **refer** macros. Macros used only at Bell Laboratories have been removed. There are a few extensions to previous **-ms** macros, and a number of new macros, but all the documented **-ms** macros still work exactly as they did before, and have the same names as before. Output produced with **-ms** should look like output produced with **-mos**

The **-ms** macros, like all **troff** macro packages, provide higher-level commands than those provided by “plain” **troff**. If you are new to the subject of macros, we suggest that you also consider the other macro packages (**-me**, included in the *bsd4.2* environment, and **-mm**, a version of which is included with the *sys5* environment; see Chapters 4 and 5 of this section). Each package has its own strengths and weaknesses. One will probably appear best-suited to your own needs.

3.2 COVER SHEETS AND FIRST PAGES

The **-ms** package has several “canned” formats, all developed at either Bell Telephone Labs or Berkeley. You may use these formats if you wish, or ask your system administrator to develop new ones that suit the needs of your site. In general, the first line of a document specifies the format of the first page. For example, if the first

line is “.RP”, the document will have a cover sheet suitable for a Bell Telephone Labs Released Paper format.

In general **-ms** is arranged so that only one form of a document need be stored, containing all information; the first command gives the format, and items unnecessary for that format are ignored.

3.3 PARAGRAPHS

The **.PP** macro should precede each paragraph. It produces indenting and extra space, and also initializes various number registers used internally by **-ms**. You should always be sure to have a **.PP** command (or its variants, such as **.LP**) near the start of your document.

If you want a non-indented paragraph, use the **.LP** macro instead. The paragraph spacing can be changed. See the comments under “Registers” below.

3.4 PAGE HEADINGS

The **-ms** macros, by default, will print a page heading containing a page number (if greater than 1). A default page footer is provided only in **nroff**, where the date is used. You can make minor adjustments to the page headings/footings by redefining the strings **LH**, **CH**, and **RH** which are the left, center and right portions of the page headings, respectively; and the strings **LF**, **CF**, and **RF**, which are the left, center and right portions of the page footer. For more complex formats, the user can redefine the macros **PT** and **BT**, which are invoked respectively at the top and bottom of each page. The margins (taken from registers **HM** and **FM** for the top and bottom margin respectively) are normally 1 inch; the page header/footer are in the middle of that space. The user who redefines these macros should be careful not to change parameters such as point size or font without later resetting them to default values.

3.5 MULTI-COLUMN FORMATS

If you place the command “.2C” in your document, the document will be printed in double column format beginning at that point. This feature is not too useful in computer terminal output, but is often desirable on the typesetter. The command “.1C” will go back to one-column format and also skip to a new page. The “.2C” command is actually a special case of the command

```
.MC [column width [gutter width]]
```

which makes multiple columns with the specified column and gutter width; as many columns as will fit across the page are used. Thus,

any reasonable number of columns-per-page can be printed. Whenever the number of columns is changed (except going from full width to some larger number of columns), a new page is started.

3.6 HEADINGS

There are two commands that produce special headings. The

`.NH`

macro produces automatically-numbered section headings (1, 2, 3, ...). The heading title will be set in boldface. For example,

`.NH`

Care and Feeding of Managers

produces

1. Care and Feeding of Managers

Alternatively,

`.SH`

Care and Feeding of Directors

will print the heading with no number added:

Care and Feeding of Directors

Every section heading, of either type, should be followed by a paragraph beginning with `.PP` or `.LP`, indicating the end of the heading. Headings may contain more than one line of text.

The `.NH` command also supports more complex numbering schemes. If a numeric argument is given, it is taken to be a "level" number and an appropriate sub-section number is generated. Larger level numbers indicate deeper sub-sections, as in this example:

`.NH`

Erie-Lackawanna

`.NH 2`

Morris and Essex Division

`.NH 3`

Gladstone Branch

`.NH 3`

Montclair Branch

`.NH 2`

Boonton Line

generates:

1. Erie-Lackawanna**1.1. Morris and Essex Division****1.1.1. Gladstone Branch****1.1.2. Montclair Branch****1.2. Boonton Line**

An explicit “.NH 0” will reset the numbering of level 1 to one, as here:

```
.NH 0
Penn Central
```

which yields

```
1. Penn Central
```

3.7 INDENTED PARAGRAPHS

The `-ms` package allows you to produce paragraphs with hanging numbers, e.g., references, descriptions, etc. The sequence

```
.IP [1]
Text for first paragraph, typed normally for as
long as you would like on as many lines as needed.
.IP [2]
Text for second paragraph, ...
```

produces this kind of output.

```
[1] Text for first paragraph, typed normally for as long as
you would like on as many lines as needed.

[2] Text for second paragraph, ...
```

A series of indented paragraphs may be followed by an ordinary paragraph beginning with `.PP` or `.LP`, depending on whether you wish indenting or not. More sophisticated uses of `.IP` are also possible. If the label is omitted, for example, a plain block indent is produced.

```
.IP
This material will
just be turned into a
block indent suitable for quotations,
callouts, and similiar things.
```

will produce

```
This material will just be turned into a block indent suitable for
quotations, call-outs, and similiar things.
```


If a nonstandard amount of indenting is required, it may be specified after the label (as an integer number of character positions) and will remain in effect until the next call of .PP or .LP. Thus, the general form of the .IP command contains two additional fields: the label and the indenting length. For example,

```
.IP first: 9
Notice the longer label, requiring larger
indenting for these paragraphs.
.IP second:
And so forth.
```

produces this:

```
first:      Notice the longer label, requiring larger indenting for
            these paragraphs.
second:     And so forth.
```

It is also possible to produce multiple nested indents; the command .RS indicates that the next .IP starts from the current indentation level. Each .RE will eat up one level of indenting so you should balance .RS and .RE commands. The .RS command should be thought of as “move right” and the .RE command as “move left”. As an example, the input below

```
.IP 1.
Text Editors
.RS
.IP 1.1.
ed
.IP 1.2.
vi
.IP 1.3.
dm editor
.RE
.IP 2.
Text Processors
.RS
.IP 2.1
troff
.IP 2.2
nroff
```

yields the following output.

1. Text Editors
 - 1.1. ed
 - 1.2. vi
 - 1.3. dm editor
2. Text Processors
 - 2.1. troff
 - 2.2. nroff

All of these variations on .LP leave the right margin untouched. Sometimes, for purposes such as setting off a quotation, a paragraph indented on both right and left is required.

A single paragraph like this is obtained by preceding it with .QP. More complicated material (several paragraphs) should be bracketed with .QS and .QE.

3.8 EMPHASIS

To get *Italics* (on the typesetter or laser printer) or underlining (on the terminal), use the .I macro, as shown below.

```
.I
as much text as you want
can be typed here
.R
```

The .R macro restores the normal (usually Roman) font. If only one word is to be *Italicized*, it may be placed on the same line as the .I command.

```
.I word
```

In this case, no .R is needed to restore the previous font. **Boldface** text can be produced by the .B macro.

```
.B
Text to be set in boldface
goes here
.R
This text is Roman.
You can also set just
.B one
word in boldface.
```

Text bolded using .B will be underlined on devices that can't produce bold type.

Type size changes can be specified with the commands .LG (make larger), .SM (make smaller), and .NL (return to normal size). The size change is two points; the commands may be repeated for cumulative effect.

If actual underlining, as opposed to Italicizing, is required on the typesetter, the command

```
.UL word
```

will underline a word.

3.9 FOOTNOTES

Material placed between lines with the commands .FS (footnote) and .FE (footnote end) will be collected, remembered, and finally placed at the bottom of the current page.

```
.FS
This is a footnote
.FE
```

By default, footnotes are 11/12th the length of normal text, but this can be changed using the FL register (see below).

Footnote numbers are printed by means of a pre-defined string (`**`), which you invoke separately from .FS and .FE. Each time it is used, this string increases the footnote number by one, whether or not you use .FS and .FE in your text. Footnote numbers will be superscripted on the phototypesetter and on daisy-wheel terminals, but on low-resolution devices (such as a lineprinter or a CRT), they will be bracketed. If you use `**` to indicate numbered footnotes, then the .FS macro will automatically include the footnote number at the bottom of the page.

Note: If you never use the "`**`" string, no footnote numbers will appear anywhere in the text, including the footnote itself. The output footnotes will look exactly like footnotes produced with `-mos`

If you are using `**` to number footnotes, but want a particular footnote to be marked with an asterisk or a dagger, then give that mark as the first argument to .FS. In the footnote, the dagger will appear where the footnote number would otherwise appear.

Footnote numbering will be temporarily suspended, because the `**` string is not used. Instead of a dagger, you could use an asterisk * or double dagger ‡, input as `\(dd`.

This is a footnote

3.10 DISPLAYS AND TABLES

To prepare displays (sections of text in which the lines should be output exactly as typed), enclose the desired lines in the commands `.DS` and `.DE`

```
.DS
Lines like these that should not be
reformatted must be placed between .DS and .DE
macros.
.DE
```

By default, lines between `.DS` and `.DE` are indented and left-adjusted. You can also center lines, or retain the left margin. Lines bracketed by `.DS C` and `.DE` commands are centered (and not rearranged); lines bracketed by `.DS L` and `.DE` are left-adjusted, not indented, and not rearranged. A plain `.DS` is equivalent to `.DS I`, which indents and left-adjusts. Thus,

```
these lines were preceded by
the .DS C command
and followed by
the .DE command;
```

whereas

```
these lines were preceded
by .DS L and followed by
the .DE command.
```

Note that `.DS C` centers each line; there is a variant `.DS B` that makes the display into a left-adjusted block of text, and then centers that entire block. Normally a display is kept together, on one page. If you wish to have a long display which may be split across page boundaries, use `.CD`, `.LD`, or `.ID` in place of the commands `.DS C`, `.DS L`, or `.DS I` respectively. An extra argument to the `.DS I` or `.DS` command is taken as an amount to indent.

3.11 BOXING WORDS OR LINES

To draw rectangular boxes around words, use the `.BX` command. The line

```
.BX word
```

will print the word “word” in a box, like this.

```
word
```

The boxes will not be neat on a terminal, and this should not be used as a substitute for Italics. Longer pieces of text may be boxed by enclosing them with `.B1` and `.B2`:

3.12 KEEPING BLOCKS TOGETHER

If you wish to keep a table or other block of lines together on a page, there are “keep - release” commands. If a block of lines preceded by `.KS` and followed by `.KE` does not fit on the remainder of the current page, a new page will be started. Lines bracketed by `.DS` and `.DE` commands are automatically kept together this way. There is also a “floating” keep. If the block to be kept together is preceded by `.KF` instead of `.KS` and does not fit on the current page, it will be moved down through the text and output at the top of the next page. Thus, no large blank space will be introduced in the document.

3.13 NROFF/TROFF REQUESTS

Among the useful requests from the basic formatting programs are the following. They all work with both typesetter and computer terminal output:

- `.bp` - begin new page
- `.br` - “break”, stop running text from line to line.
- `.sp n` - insert *n* blank lines.
- `.na` - don't adjust right margins.

3.14 DATE

By default, documents produced on computer terminals have the date at the bottom of each page; documents produced on the typesetter don't. To force the date, use `“.DA”`. To force no date, use `“.ND”`. To set an explicit date, add an argument to `.DA`. for example

```
.DA July 4, 1776
```

This puts the date “July 4, 1776” at the bottom of each page.

3.15 REGISTERS

Certain of the registers used by `-ms` can be altered to change default settings. They should be changed with `.nr` commands, as with

```
.nr PS 9
```

to make the default point size 9 point. If the effect is needed immediately, the normal `troff` command should be used in addition to changing the number register.

The table below is a summary of `-ms` number registers.

Register	Defines	Takes effect	Default
PS	point size	next para.	10
VS	line spacing	next para.	12 pts
LL	line length	next para.	6' '
LT	title length	next para.	6' '
PD	para. spacing	next para.	0.3 VS
PI	para. indent	next para.	5 ens
FL	footnote length	next FS	11/12 LL
CW	column width	next 2C	7/15 LL
GW	intercolumn gap	next 2C	1/15 LL
PO	page offset	next page	26/27' '
HM	top margin	next page	1' '
FM	bottom margin	next page	1' '

You may also alter the strings LH, CH, and RH which are the left, center, and right headings respectively; and similarly LF, CF, and RF which are strings in the page footer. The page number on output is taken from register PN, to permit changing its output style. For more complicated headers and footers the macros PT and BT can be redefined, as explained earlier.

3.16 ACCENTS

To simplify typing certain foreign words, strings representing common accent marks are defined. There are two types of accent marks. The “old” marks, used in the **-mos** package, precede the letter over which the mark is to appear. The newer (*bsd4.2*) marks follow the letter. The “old” (*sys5*) marks are listed below.

Name	Input	Output
acute accent	*' e	' e
grave accent	*' e	` e
umlaut	*:u	¨ u
circumflex	*^e	ê
tilde	*~n	ñ
hacek	*Ce	č
cedilla	*,c	, c

The new string definitions (enhanced at Berkeley) define quotation marks and dashes for **nroff** and **troff**. The *- string will yield two hyphens in **nroff**, but in **troff** it will produce an em dash. The *Q and *U strings will produce “ and ” in **troff**, but ” in **nroff**.

These strings also add a number of foreign accent marks. All the older accent marks still work. However, by placing .AM at the beginning of your document, you can use the enhanced accent marks instead. Unlike the older **-mos** accent marks, the accent

strings should come after the letter being accented. Here is a list of the diacritical marks, with examples of what they look like.

Name	Input	Output
acute accent	e\ <i>'</i>	é
grave accent	e\ <i>`</i>	è
circumflex	o\ <i>^</i>	ô
cedilla	c\ <i>,</i>	ç
tilde	n\ <i>~</i>	ñ
question	*? <i></i>	¿
exclamation	*! <i></i>	¡
umlaut	u\ <i>:</i>	ü
digraph s	*8 <i></i>	ß
hacek	c\ <i>v</i>	č
macron	a\ <i>_</i>	ā
underdot	s\ <i>.</i>	š
o-slash	o\ <i>/</i>	ó
angstrom	a\ <i>o</i>	Å
yogh	kni\ <i>*3t</i>	knjȳ
Thorn	*(Th <i></i>	Þ
thorn	*(th <i></i>	þ
Eth	*(D- <i></i>	Ð
eth	*(d- <i></i>	ð
hooked o	*q <i></i>	ø
ae ligature	*(ae <i></i>	æ
AE ligature	*(Ae <i></i>	Æ
oe ligature	*(oe <i></i>	œ
OE ligature	*(Oe <i></i>	Œ

To use these new diacritical marks, include the macro

`.AM`

at the beginning of your file. Without it, some will not print at all, and others will be placed on the wrong letter.

3.17 PRINTING

After your document is prepared and stored on a file, you can print it on a terminal with the command

`nroff -ms file`

and you can print it on the typesetter with the command

`troff -ms file`

Many command line options are possible. See the information on `troff` in Chapters 1 and 2 of this section.

In each case, if your document is stored in several files, you may list all the filenames where we have used *file*. If equations or tables are used, `eqn` and/or `tbl` must be invoked as preprocessors.

Note: If `.2C` was used to produce two-column text, pipe the `nroff` output through `col` before viewing it on an edit/transcript pad or printing it. You may do this on the command line, by using the pipe symbol

```
nroff -ms file col
```

or, if you always want the output piped through `col`, you may make the first line of the file

```
.pi /bin/col
```

3.18 TYPESETTING MATHEMATICS

If you have to typeset equations or Greek characters, see the information on `eqn` in Chapter 4 of this section. To aid `eqn`, `-ms` provides definitions of `.EQ` and `.EN` that normally center the equation and set it off slightly from the surrounding text. An argument to `.EQ` is taken to be an equation number and is placed in the right margin near the equation. In addition, there are three special arguments to `.EQ`: the letters C, I, and L indicate centered (default), indented, and left adjusted equations, respectively. If there is both a format argument and an equation number, give the format argument first, as in

```
.EQ L (1.3a)
```

for a left-adjusted equation numbered (1.3a).

Similarly, the macros `.TS` and `.TE` are defined to separate tables from text with a little space. A very long table with a heading may be broken across pages by beginning it with `.TS H` instead of `.TS`, and placing the line `.TH` in the table data after the heading. If the table has no heading repeated from page to page, just use the ordinary `.TS` and `.TE` macros.

3.19 BIBLIOGRAPHY ENTRIES

The `bsd4.2` version of `-ms` includes a macro especially for bibliography entries, called `.XP`, which stands for exdented paragraph. It will exdent the first line of the paragraph by `\n`(PI units, usually 5n (the same as the indent for the first line of a `.PP`). Most bibliographies are printed this way. Of course, you will have to take care of italicizing the book title and journal, and quoting the title of the journal article. Indentation or exdentation can be changed by setting the value of number register PI.

If you need to produce endnotes rather than footnotes, put the references in a file of their own. This is similar to what you would do if you were typing the paper on a conventional typewriter. Note that you can use automatic footnote numbering without actually

having .FS and .FE pairs in your text. If you place footnotes in a separate file, you can use .IP macros with ** as a hanging tag; this will give you numbers at the left-hand margin. With some styles of endnotes, you would want to use .PP rather than .IP macros, and specify ** before the reference begins.

3.20 TABLE OF CONTENTS

There are four new macros to help produce a table of contents. Table of contents entries must be enclosed in .XS and .XE pairs, with optional .XA macros for additional entries; arguments to .XS and .XA specify the page number, to be printed at the right. A final .PX macro prints out the table of contents. The .XS and .XE pairs may also be used in the text, after a section header for instance, in which case page numbers are supplied automatically. However, most documents that require a table of contents are too long to produce in one run, which is necessary if this method is to work. It is recommended that you do a table of contents after finishing your document. To print out the table of contents, use the .PX macro; if you forget it, nothing will happen.

It is also possible to produce custom headers and footers that are different on even and odd pages. The .OH and .EH macros define odd and even headers, while .OF and .EF define odd and even footers. Arguments to these four macros are specified as with .tl. Note that it would be an error to have an apostrophe in the header text; if you need one, you will have to use a different delimiter around the left, center, and right portions of the title. You can use any character as a delimiter, provided it doesn't appear elsewhere in the argument to .OH, .EH, .OF, or EF.

Both versions of **-ms** work in conjunction with the **tbl**, and **eqn**. Only the "new" version works well with the **refer** preprocessor.

Note: In the **bsd4.2** version of **ms**, macros to deal with tables, equations, and bibliography citations are read in only as needed, as are the thesis macros (.TM), the special accent mark definitions (.AM), table of contents macros (.XS and .XE), and macros to format the optional cover page. The code for the **-ms** package lives in */usr/lib/tmac/tmac.s*, and sourced files reside in the

directory */usr/lib/ms*.

3.21 SUMMARY OF *-ms* MACROS

Macros marked with a † exist in *-mos* only. Macros marked with a ‡ exist in *-ms* only.

<i>Name</i>	<i>Function</i>	<i>Name</i>	<i>Function</i>
1C	One-column format	LG	Increase type size
2C	Two-column format	LP	Left-aligned paragraph
AB	Begin abstract	ND	Change or reset date
AE	End abstract	NH	Specify numbered heading
AI	Specify author's institution	NL	Use normal type size
AU	Specify author	PP	Begin paragraph
B	Begin boldface	R	Return to regular font
DA	Print date on each page	RE	To previous relative indent
DE	End display	RP †	BTL released paper format
DS	(CD, LD, ID) Start display	RS	Next relative indent level
EN	End equation	SH	Specify section heading
EQ	Begin equation	TL	Specify title
FE	End footnote	UL	Underline one word
FS	Begin footnote		
SG	Insert signature line		
I	Begin Italics		
IX ‡	Index Entry		
SM	Change to smaller type size		
IP	Begin indented paragraph		
KE	Release keep		
KF	Begin floating keep		
KS	Start keep		

3.22 SUMMARY OF *-ms* REGISTER NAMES

The following register names are used by *-ms* internally. Independent use of these names in one's own macros may produce incorrect output. Note that no lowercase letters are used in any *-ms* internal name.

Number registers used in <i>-ms</i>										
:	DW	GW	HM	IQ	LL	NA	OJ	PO	T.	TV
#T	EF	H1	HT	IR	LT	NC	PD	PQ	TB	VS
1T	FL	H3	IK	KI	MM	NF	PF	PX	TD	YE
AV	FM	H4	IM	L1	MN	NS	PI	RO	TN	YY
CW	FP	H5	IP	LE	MO	OI	PN	ST	TQ	ZN

String registers used in <code>-ms</code>										
'	A5	CB	DW	EZ	I	KF	MR	R1	RT	TL
`	AB	CC	DY	FA	I1	KQ	ND	R2	S0	TM
^	AE	CD	E1	FE	I2	KS	NH	R3	S1	TQ
~	AI	CF	E2	FJ	I3	LB	NL	R4	S2	TS
:	AU	CH	E3	FK	I4	LD	NP	R5	SG	TT
,	B	CM	E4	FN	I5	LG	OD	RC	SH	UL
1C	BG	CS	E5	FO	ID	LP	OK	RE	SM	WB
2C	BT	CT	EE	FQ	IE	ME	PP	RF	SN	WH
A1	C	D	EL	FS	IM	MF	PT	RH	SY	WT
A2	C1	DA	EM	FV	IP	MH	PY	RP	TA	XD
A3	C2	DE	EN	FY	IZ	MN	QF	RQ	TE	XF
A4	CA	DS	EQ	HO	KE	MO	R	RS	TH	XK

C

C

C

C

C

Chapter 4: The -me Macro Package

4.1 INTRODUCTION

The **-me** macro package, developed at Berkeley by Eric Allman and others, has several unique features, the most obvious of which is its use of lower-case characters in macro names. There are also a number of macro parameters that may be adjusted.

Note: The **-me** macros are only available in the *bsd4.2* version of DOMAIN/IX.

In **-me**, fonts may be set to a font number only. In **nroff**, font 8 is underlined, and is set in bold font in **troff** (although font 3, bold in **troff**, is not underlined in **nroff**). Font 0 is no font change; the font of the surrounding text is used instead. Notice that fonts 0 and 8 are pseudo-fonts; that is, they are simulated by the macros. This means that although it is legal to set a font register to zero or eight, it is not legal to use the escape character form (e.g. `\f8`).

All distances are specified in basic units, so it is nearly always necessary to use a scaling factor. For example, the request to set the paragraph indent to eight one-en spaces is

```
.nr pi 8n
```

and not

```
.nr pi 8
```

which would set the paragraph indent to eight basic units, or about 0.02 inch. Default parameter values are given in brackets in the remainder of this section.

Registers and strings of the form `$x` may be used in expressions but should not be changed. Macros of the form `$x` perform some function (as described) and may be redefined to change this function.

All names in **-me** follow a rigid naming convention. You may define number registers, strings, and macros, provided that you use single-character uppercase names or double-character names consisting of letters and digits, with at least one uppercase letter. In no case should special characters be used in user-defined names.

On daisy wheel type printers in twelve pitch, the **-rx1** flag can be set to make lines default to one-eighth inch (the normal spacing for a newline in twelve-pitch). This is normally too small for easy readability, so the default is to space one sixth inch.

4.2 PARAGRAPHING

These macros are used to begin paragraphs. The standard paragraph macro is **.pp**. The others are all variants to be used for special purposes.

Note: The first call to one of the paragraphing macros defined in this section (or to the **.sh** macro defined in the next section) *initializes* the macro processor. After initialization, you may not use any of the following requests.

.sc
.lo
.th
.ac

You should also avoid changing parameters that have a global effect on the format of the page (e.g., page length and header/footer margins).

- .lp** Begin left-justified paragraph. Centering and underlining are turned off if they were on, the font is set to $\backslash(\mathbf{pp}[1]$ and the type size is set to $\backslash(\mathbf{ps}[10\mathbf{p}]$. A $\backslash(\mathbf{ps}$ space is inserted before the paragraph [0.35v in **troff**, 1v or 0.5v in **nroff** depending on device resolution]. The indent is reset to $\backslash(\mathbf{i}[0]$ plus $\backslash(\mathbf{po}[0]$ unless the paragraph is inside a display. (see **.ba**). At least the first two lines of the paragraph are kept together on a page.
- .pp** Like **.lp**, except that it uses $\backslash(\mathbf{pi}[5\mathbf{n}]$ units of indent. This is the standard paragraph macro.
- .ip T I** Indented paragraph with hanging tag. The body of the following paragraph is indented *I* spaces (or $\backslash(\mathbf{n(ii}[5\mathbf{n}]$ spaces if *I* is not specified) more than a non-indented paragraph (such as with **.P**) is. The title *T* is exdented (opposite of indented). The result is a paragraph with an even left edge and *T* printed in the margin. Any spaces in *T* must be unpaddable. If *T* will not fit in the space provided, **.ip** will start a new line.
- .np** A variant of **.ip** that numbers paragraphs. Numbering is reset after a **.lp**, **.P**, or **.sh**. The current paragraph number is in $\backslash(\mathbf{n}(\$p$.

4.3 SECTION HEADINGS

Numbered sections are similiar to paragraphs except that a section number is automatically generated for each one. The section

numbers are of the form

1.2.3.

The depth of the section is the count of numbers (separated by decimal points) in the section number. Unnumbered section headings are similar, except that no number is attached to the heading.

.sh +N T a b c d e f Begin numbered section of depth *N*. If *N* is missing, the current depth (maintained in the number register `\n($0)` is used. The values of the individual parts of the section number are maintained in `\n($1` through `\n($6`. There is a `\n(ss[1v]` space before the section. *T* is printed as a section title in font `\n(sf[8]` and size `\n(sp [10p]`. The “name” of the section may be accessed via `\n($n`. If `\n(si` is non-zero, the base indent is set to `\n(si` times the section depth, and the section title is extended. (See **.ba**.) Also, an additional indent of `\n(so[0]` is added to the section title (but not to the body of the section). The font is then set to the paragraph font, so that more information may occur on the line with the section number and title. The **.sh** macro ensures that there is enough room to print the section head plus the beginning of a paragraph (about 3 lines total). If arguments *a* through *f* are specified, the section number is set to that number rather than incremented automatically. If any of arguments *a* through *f* is a hyphen, that number is not reset. If *T* is a single underscore, then the section depth and numbering is reset, but the base indent is not reset and nothing is printed out. This is useful to automatically coordinate section numbers with chapter numbers.

.sx +N Go to section depth *N*[-1] but do not print the number and title, and do not increment the section number at level *N*. This has the effect of starting a new paragraph at level *N*.

.uh T Unnumbered section heading. The title *T* is printed with the same rules for spacing, font, etc., as are used by **.sh**.

- .\$p *T B N*** Print section heading. These may be redefined to get fancier headings. *T* is the title provided on the **.sh** or **.uh** line; *B* is the section number for this section, and *N* is the depth of this section. These parameters are not always present; in particular, **.sh** passes all three, while **.uh** passes only the first. The **.sx** macro passes all three, but the first two are null strings. Care should be taken if this macro is redefined; it is quite complex and subtle.
- .\$0 *T B I*** This macro is called automatically after every call to **.\$p**. It is normally undefined, but may be used to automatically put every section title into the table of contents or for some similar function. *T* is the section title for the section title which was just printed *B* is the section number. *N* is the section depth.
- .\$1 - \$6** These are traps called just before printing the section of depth **\$n**. May be defined to (for example) give variable spacing before sections. These macros are called from **.\$p**, so if you redefine that macro you may lose this feature.

4.4 HEADERS AND FOOTERS

Headers and footers are put at the top and bottom of every page automatically. They are set in font **\n(tf[3]** and size **\n(tp[10p]**. Each of the definitions apply as of the next page. Three-part titles must be quoted if there are two blanks adjacent anywhere in the title or more than eight blanks total.

The spacing of headers and footers is controlled by three number registers. **\n(hm[4v]** is the distance from the top of the page to the top of the header, **\n(fm[3v]** is the distance from the bottom of the page to the bottom of the footer, **\n(tm[7v]** is the distance from the top of the page to the top of the text, and **\n(bm[6v]** is the distance from the bottom of the page to the bottom of the text (nominal). The macros **m1** through **m4** are also supplied for compatibility with ROFF documents.

- he 'l' m' r'** Define three-part header, to be printed on the top of every page.
- fo 'l' m' r'** Define footer, to be printed at the bottom of every page.

eh 'l'm'r'	Define header, to be printed at the top of every even-numbered page.
oh 'l'm'r'	Define header to be printed at the top of every odd-numbered page.
ef 'l'm'r'	Define footer, to be printed at the bottom of every even-numbered page.
of 'l'm'r'	Define footer to be printed at the bottom of every odd-numbered page.
hx	Suppress headers and footers on the next page.
m1 +N	Set the space between the top of the page and the header [4v].
m2 +N	Set the space between the header and the first line of text [2v].
m3 +N	Set the space between the bottom of the text and the footer [2v].
m4 +N	Set the space between the footer and the bottom of the page [4v].
ep	End this page, but do not begin the next page. Useful for forcing out footnotes, but otherwise seldom used. Must be followed by .bp or the end of input.
\$h	Called at every page to print the header. May be redefined to provide fancy (e.g., multi-line) headers, but doing so loses the function of the he , fo , eh , oh , ef , and of requests, as well as the chapter-style title feature of +c .
.\$f	Print footer; same comments apply as in \$h .
\$H	A normally undefined macro which is called at the top of each page (after outputting the header, initial saved floating keeps, etc.); in other words, this macro is called immediately before printing text on a page. It can be used for column headings and the like.

4.5 DISPLAYS

All displays except centered blocks and block quotes are preceded and followed by an extra **\n(bs** (same as **\n(ps**) space. Quote spacing is stored in a separate register; centered blocks have no default initial or trailing space. The vertical spacing of all displays except quotes and centered blocks is stored in register **\n(\$R** instead of **\n(\$r**.

- .(l m f** Begin list. Lists are single-spaced, unfilled text. If *f* is **F**, the list will be filled. If *m* is **I**, the list will be indented by `\n(bi[4n]`; if *m* is **M** the list will be indented to the left margin; if *m* is **L** the list is left-justified with respect to the text (different from **M** only if the base indent (stored in `\n($i` and set with `.ba`) is not zero); and if *m* is **C** the list will be centered on a line-by-line basis. The list is set in font `\n(df [0]`. Must be matched by a `.)l`. This macro is almost like `.B` except that no attempt is made to keep the display on one page.
- .)l** End list.
- .(q** Begin major quote. Major quotes are single spaced, filled, moved in from the text on both sides by `\n(qi[4n]`, preceded and followed by `\n(qs[same as \n(bs]` space, and are set in point size `\n(qp [one point smaller than surrounding text]`.
- .)q** End major quote.
- .B m f** Begin block. Blocks are a form of “keep,” where the text of a keep is kept together on one page if possible (keeps are useful for tables and figures which should not be broken over a page). If the block will not fit on the current page, a new page begins, unless that would leave more than `\n(bt[0]` white space at the bottom of the text. If `\n(bt` is zero, the threshold feature is turned off. Blocks are not filled unless *f* is **F**. A block will be left-justified if *m* is **L**, indented by `\n(bi[4n]` if *m* is **I** or absent, centered (line-for-line) if *m* is **C**, and left-justified to the margin (not to the base indent) if *m* is **M**. The block is set in font `\n(df [0]`.
- .R** End block.
- .(z m f** Begin floating keep. Like `.B`, except that the keep is *float*ed to the bottom of the page or the top of the next page. Therefore, its position relative to the text changes. The floating keep is preceded and followed by `\n(zs[1v]` space. Also, it defaults to mode **M**.
- .)z** End floating keep.
- .(c** Begin centered block. The next keep is centered as a block, rather than on a line-by-line

basis as with **.(b C** This call may be nested inside keeps.

.)c End centered block.

4.6 ANNOTATIONS

.(d Begin delayed text. Everything in the next keep is saved for output later with **.pd**, in a manner similar to footnotes.

.)d n End delayed text. The delayed text number register **\n(\$d** and the associated string ***#** are incremented if ***#** has been referenced.

.pd Print delayed text. Everything diverted via **.(d** is printed and truncated. This might be used at the end of each chapter to produce endnotes.

.(f Begin footnote. The text of the footnote is floated to the bottom of the page and set in font **\n(ff[1]** and size **\n(fp[8p]**. Each entry is preceded by **\n(fs[0.2v]** space, is indented **\n(fi[3n]** on the first line, and is indented **\n(fu[0]** from the right margin. Footnotes line up underneath two-column output. If the text of the footnote will not all fit on one page, it will be carried over to the next page.

.)f n End footnote. The number register **\n(\$f** and the associated string ***#** are incremented if they have been referenced.

.\$s The macro to output the footnote separator. This macro may be redefined to give other size lines or other types of separators. Normally, it draws a 1.5i line.

.(x x Begin index entry. Index entries are saved in the index **x [x]** until called up with **.xp**. Each entry is preceded by a **\n(xs[0.2v]** space. Each entry is exdented by **\n(xu[0.5i]**; this register tells how far the page number extends into the right margin.

.)x P A End index entry. The index entry is finished with a row of dots with **A [null]** right-justified on the last line (such as for an author's name), followed by **P [\n%]**. If **A** is specified, **P** must also be specified; **\n%** can be used to print the current page number. If **P** is an underscore, no page number and no row of dots will be

printed.

.xp x Print index x [x]. The index is formatted in the font, size, and so forth in effect at the time it is printed, rather than at the time entries were collected.

4.7 MULTI-COLUMN OUTPUT

.2c $+S N$ Enter two-column mode. The column separation is set to $+S$ [$4n, 0.5i$ in ACM mode] (saved in $\backslash n(\$s)$). The column width, calculated to fill the single-column line length with both columns, is stored in $\backslash n(\$l)$. The current column is in $\backslash n(\$c)$. You can test register $\backslash n(\$m[1])$ to see whether you are in single-column or double-column mode. Actually, the request enters N [2] columned output.

.1c Revert to single-column mode.

.bc Begin column. This is like **.bp** except that it begins a new column on a new page only if necessary, rather than forcing a whole new page if there is another column left on the current page.

4.8 FONTS AND SIZES

.sz $+P$ The pointsize is set to P [$10p$], and the line spacing is set proportionally. The ratio of line spacing to pointsize is stored in $\backslash n(\$r)$. The ratio used internally by displays and annotations is stored in $\backslash n(\$R)$ (although this is not used by **.sz**).

.r $W X$ Set W in roman font, appending X in the previous font. To append different font requests, use $X = \backslash c$. If no parameters, change to roman font.

.i $W X$ Set W in Italics, appending X in the previous font. If no parameters, change to italic font. Underlines in **nroff**.

.b $W X$ Set W in bold font and append X in the previous font. If no parameters, switch to bold font. Underlines in **nroff**.

.rb $W X$ Set W in bold font and append X in the previous font. If no parameters, switch to bold font. **.rb** differs from **.b** in that **.rb** does not

underline in **nroff**.

.u *W X*

Underline *W* and append *X*. This requests true underlining, as opposed to **.ul**, which changes to the “underline” font (usually italics in **troff**). It won’t work right if *W* is spread or broken (including hyphenated). In other words, it is safe in nofill mode only.

.q *W X*

Quote *W* and append *X*. In **nroff**, this surrounds *W* with double quote marks. In **troff**, it uses properly-directed (opening and closing) quotes.

.bi *W X*

Set *W* in bold italics and append *X*. Actually, sets *W* in italic and overstrikes once. Underlines in **nroff**. It won’t work right if *W* is spread or broken (including hyphenated). In other words, it is safe in nofill mode only.

.bx *W X*

Sets *W* in a box with *X* appended. Underlines in **nroff**. It won’t work right if *W* is spread or broken (including hyphenated). In other words, it is safe in nofill mode only.

4.9 ROFF SUPPORT

.ix *+N*

Indent, no break. Equivalent to ‘**in** *N*’.

.bl *N*

Leave *N* units of contiguous white space. Use on the next page if not enough room on this page. Equivalent to a **.sp** *N* inside a block.

.pa *+N*

Equivalent to **.bp**.

.ro

Set page number in Roman numerals. Equivalent to **.af** % **i**.

.ro

Set page number in arabic. Equivalent to **.af** % **1**.

.n1

Number lines, placing line numbers in margin.

.n2 *N*

Number lines from *N*, stop if *N* = 0.

.sk

Leave the next output page blank, except for headers and footers. This is used to leave space for a full-page diagram to be pasted in later. To get a partial-page paste-in display, say **.sv** *N*, where *N* is the amount of space to leave; this space will be output immediately if there is room, and will otherwise be output at the top of the next page. If *N* is greater than the amount of available space on an empty

page, no space will ever be output.

4.10 PREPROCESSOR SUPPORT

- .EQ** *m t* Begin equation. The equation is centered if *m* is **C** or omitted, indented $\backslash\mathbf{n}(\mathbf{bi}[4\mathbf{n}]$ if *m* is **I**, and left justified if *m* is **L**. *T* is a title printed on the right margin next to the equation.
- .EN** *c* End equation. If *c* is **C**, the equation must be continued by immediately following with another **.EQ**, the text of which can be centered along with this one. Otherwise, the equation is printed, always on one page, with $\backslash\mathbf{n}(\mathbf{es} [0.5\mathbf{v}$ in **troff**, $1\mathbf{v}$ in **nroff**] space above and below it.
- .TS** *h* Table start. Tables are single-spaced and kept on one page if possible. If you have a table that will not fit on one page, set *h* to **H** and follow the header part (to be printed on every page of the table) with a **.TH**.
- .TH** With ends the header portion of the table (started by **.TS H**.)
- .TE** Table end. Note that this table does not float, in fact, it is not even guaranteed to stay on one page if you use requests such as **.sp** intermixed with the text of the table. If you want it to float (or if you use requests inside the table), surround the entire table (including the **.TS** and **.TE** requests) with the requests **.(z and .)z**.

4.11 MISCELLANEOUS MACROS

- .re** Reset tabs. Tabs are normally set to every 0.5i in **troff** and every **0.8i** in **nroff**.
- .ba** *+N* Set the base indent to *+N* [0] (saved in $\backslash\mathbf{n}(\mathbf{\$i}$). All paragraphs, sections, and displays come out indented by this amount. Titles and footnotes are unaffected. The **.sh** request performs a **.ba** request if $\backslash\mathbf{n}(\mathbf{si}[0]$ is not zero, and sets the base indent to $\backslash\mathbf{n}(\mathbf{si}*\backslash\mathbf{n}(\mathbf{\$0}$.
- .xl** *+N* Set the line length to *N* [6.0i]. This differs from **.ll** in that it only affects the current environment.
- .ll** *+N* Set line length in all environments to *N* [6.0i]. This should not be used after output has

begun, especially two-column output. The current line length is stored in `\n($l`.

- .hl** Draws a horizontal line the length of the page. This is useful inside floating keeps to differentiate between the text and the figure.
- .lo** This macro loads those macros that reside in `/usr/lib/me/local.me`. These locally-defined macros should all be of the form `.*X`, where `X` is any letter (upper or lower case) or digit.

4.12 STANDARD PAPERS

- .tp** Begin title page. Spacing at the top of the page can occur. Headers and footers are suppressed. The page number is not incremented for this page.
- .th** Set thesis mode. This defines the modes acceptable for a doctoral dissertation at Berkeley. It double spaces, defines the header to be a single page number, and changes the margins to be 1.5 inch on the left and one inch on the top. `.++` and `.+c` should be used with it. This macro must be stated before initialization, that is, before the first call to a paragraphing or `.sh` macro.
- .++ m H** This request defines the section of the paper. The section type is defined by `m`. `C` signifies the chapter portion of the paper, `A` signifies the appendix portion of the paper, and `P` means that the material following should be the preliminary portion (abstract, table of contents, etc.) of the paper. `AB` signifies the abstract (numbered independently from 1 in Arabic numerals), and `B` signifies the bibliographic portion at the end of the paper. The variants `RC` and `RA` specify renumbering of pages from 1 at the beginning of each chapter or appendix, respectively. The `H` parameter defines the new header. If there are any spaces in it, the entire header must be quoted. If you want the header to have the chapter number in it, use the string `\\\\n(ch`. For example, to number appendices `A.1` etc., type `.++ RA ' ' '\\n(ch.%'`. Each section (chapter, appendix, etc.) should be preceded by the `.+c` request. It is easier when using `troff` to put

the front material at the end of the paper, so that the table of contents can be collected and output. This material can then be physically moved to the beginning of the paper as necessary.

- .+c T** Begin chapter with title *T*. The chapter number is maintained in `\n(ch`. This register is incremented every time `.+c` is called with a parameter. The title and chapter number are printed by `.$c`. The header is moved to the footer on the first page of each chapter. If *T* is omitted, `.$c` is not called. This is useful for doing your own title page at the beginning of a paper without a proper title page. `.$c` calls `.$C` as a hook so that chapter titles can be inserted into a table of contents automatically. The footnote numbering is reset to one.
- .\$c T** Print chapter number (from `\n(ch`) and *T*. This macro can be redefined to your liking. This macro calls `.$C`, which can be defined to make index entries and similar items.
- .\$C K N T** This macro is called by `.$c`. It is normally undefined, but can be used to automatically insert things like index entries. *K* is a keyword, either "Chapter" or "Appendix" (depending on the `.B .++` mode). *N* is the chapter or appendix number, and *T* is the chapter or appendix title.
- .ac A N** This macro (short for `.acm`) sets up the `nroff` environment for photo-ready papers as used by the ACM (Association for Computing Machinery). This format is 25% larger, and has no headers or footers. The author's name *A* is printed at the bottom of the page (but off the part which will be printed in the conference proceedings), together with the current page number and the total number of pages *N*. In addition, this macro loads the file `/usr/lib/me/acm.me`, which may later be augmented with other macros useful for printing papers for ACM conferences. It should be noted that this macro will not work correctly in `troff`, since it sets the page length wider than the physical width of the phototypesetter roll.

4.13 PRE-DEFINED STRINGS

<code>**</code>	Footnote number, actually <code>*[\backslashn(\f\backslash$*)</code> This macro is incremented after each call to <code>.)f</code> .
<code>*#</code>	Delayed text number. Actually <code>\n(\$d]</code> .
<code>*[</code>	Superscript. This string gives upward movement and a change to a smaller point size if possible. Otherwise it prints the left bracket character. Extra space is left above the line to allow room for the superscript.
<code>*]</code>	End superscript. Inverse to <code>*[</code> .
<code>*<</code>	Subscript. Defaults to ' <code><</code> ' if half-linefeed (and reverse half-linefeed) not possible. Extra space is left below the line to allow for the subscript.
<code>*></code>	Inverse to <code>*<</code> .
<code>*(dw</code>	The day of the week, as a word.
<code>*(mo</code>	The month, as a word.
<code>*(td</code>	Today's date, directly printable. The date is of the form December 27, 1984 . Other forms of the date can be used by using <code>\n(dy</code> (the day of the month; for example, 26), <code>*(mo</code> (as noted above), <code>\n(mo</code> (the same, but as an ordinal number; for example, December is 12). and <code>\n(yr</code> (the last two digits of the current year).
<code>*(lq</code>	Left quote marks. Double quote in nroff .
<code>*(rq</code>	Right quote.
<code>*-</code>	Three-quarter em dash in troff ; two hyphens in nroff .

4.14 SPECIAL CHARACTERS AND MARKS

There are a number of special characters and diacritical marks (such as accents) available through **-me**. To reference these characters, you must call the macro **.sc** to define the characters before using them.

.sc	Define special characters and diacritical marks, as described in the Appendix "Writing Papers With -me" at the end of this <i>Text Processing Guide</i> . This macro must be stated before initialization.
------------	--

C

C

C

C

C

Chapter 5: The -mm Macro Package

5.1 INTRODUCTION

The **-mm** macros included with the *sys5* version of DOMAIN/IX are version 15.110 — the version supplied with UNIX System III Later versions of **-mm** have been unbundled from UNIX System V and incorporated into the *Documenter's Workbench*, which is sold under separate license. The **-mm** macros combine some of the features found in **-ms** and **-me** with a number of unique capabilities that make these macros especially easy to use and modify. The uses of **-mm** range from single-page letters to documents of several hundred pages in length, such as user guides, design proposals, etc. This chapter is based on the original Bell Labs **-mm** papers by D. W. Smith, J. R. Mashey, and others.

Note: The **-mm** macros are only available in the *sys5* version of DOMAIN/IX.

5.2 CONVENTIONS

Each section of this chapter explains a single facility of **-mm**. In general, the earlier a section occurs, the more important it is for most users to read. Some of the later sections can be completely ignored if **-mm** defaults are acceptable. Likewise, each section progresses from normal-case to special-case facilities. We recommend reading a section in detail only until you have enough information to obtain the desired effect, then skimming the rest of it, since later details may be of use to comparatively few people.

The examples of output in this manual are as produced by **troff**; **nroff** output would, of course, look somewhat different. In those cases where the behavior of the two formatters is truly different, the **nroff** action is described first, with the **troff** action following in parentheses. For example:

The title is underlined (*Italic*).

means that the title is underlined in **nroff** and *Italic* in **troff**.

5.3 INPUT FILE STRUCTURE

The input for a document that is to be formatted with **-mm** includes up to four major segments, any of which may be omitted; if present, they must occur in the following order:

- **Preamble**—This segment sets the general style and appearance of a document. You can control page width, margin justification, numbering styles for headings and lists, page

headers and footers, and many other properties of the document. You can also add macros or redefine existing ones. You can omit this segment entirely if you are satisfied with default values. It produces no actual output, but only performs the setup for the rest of the document.

- *Beginning*—This segment includes those items that occur only once, at the beginning of a document, e.g., title, author's name, date.
- *Body*—This segment is the actual text of the document. It may be as small as a single paragraph, or as large as hundreds of pages. It may have a hierarchy of headings up to seven levels deep. Headings are automatically numbered (if desired) and can be saved to generate the table of contents. Five additional levels of subordination are provided by a set of list macros for automatic numbering, alphabetic sequencing, and “marking” of list items. The body may also contain various types of displays, tables, figures, references, and footnotes.
- *Ending*—This segment contains those items that occur once only, at the end of a document (e.g., signatures, lists of notations, etc.). Certain macros may be invoked here to print information that is wholly or partially derived from the rest of the document, such as the table of contents or the cover sheet.

The existence and size of these four segments varies widely among different document types. Although a specific item (such as date, title, author name(s), etc.) may be printed in several different ways depending on the document type, there is a uniform way of typing it in.

5.4 FORMATTERS

The term formatter refers to either of the text-formatting programs **nroff** and **troff**.

Requests are built-in commands recognized by the formatters. Although you rarely will need to use these requests directly, this chapter contains references to some of them. For example, the request:

```
.sp
```

inserts a blank line in the output.

Macros are named collections of requests. Each macro is an abbreviation for a collection of requests. **-mm** supplies many macros, and allows you to define additional ones. Macros and requests share the same set of names and are used in the same way.

Strings provide character variables, each of which names a string of characters. Strings are often used in page headers, page footers, and lists. They share the pool of names used by requests and

macros. A string can be given a value via the `.ds` (define string) request, and its value can be obtained by referencing its name, preceded by “*” (for 1-character names) or “*(” (for 2-character names). For instance, the string *DT* in `-mm` normally contains the current date, so that the input line:

```
Today is \*(DT.
```

resulted, today, in the following output:

```
Today is May 28, 1985.
```

The current date can be replaced, e.g.:

```
.ds DT 05/01/82
```

or by invoking a macro designed for that purpose.

Number registers fill the role of integer variables. They are used for flags, for arithmetic, and for automatic numbering. A number register can be given a value using a `.nr` request, and may be referenced by preceding its name by “\n” (for 1-character names) or “\n(” (for 2-character names). For example, the following sets the value of the register *d* to 1 more than that of the register *dd*:

```
.nr d 1+\n(dd
```

For more details on **troff**, its requests, and its various registers, see Chapters 1 and 2 of this section.

5.5 INVOKING THE MACROS

This section explains how to access `-mm`, shows UNIX command lines appropriate for various output devices, and describes command-line flags for `-mm`.

5.5.1 The mm Command

The `mm[1]` command can be used to print documents using **nroff** and `-mm`. This command invokes **nroff** with the `-cm` flag. It has options to specify preprocessing by `tbl[1]` and/or by `neqn[1]`, and for postprocessing by various output filters. Any arguments or flags that are not recognized by `mm[1]`, are passed to **nroff** or to `-mm`, as appropriate. The following options can occur in any order, although they must appear before the file name(s).

- e **neqn[1]** is to be invoked.
- t **tbl[1]** is to be invoked.
- c **col[1]** is to be invoked.
- E the “-e” option of **nroff** is to be invoked.
- y `-mm` (uncompacted macros) is to be used instead of `-cm`.
- 12 12-pitch mode is to be used. Be sure that the pitch switch on the terminal is set to 12.
- T450 output is to a DASI450. This is the default

	terminal type (unless \$TERM is set). It is also equivalent to -T1620.
-T450-12	output is to a DASI450 in 12-pitch mode.
-T300	output is to a DASI300 terminal.
-T300-12	output is to a DASI300 in 12-pitch mode.
-T300S	output is to a DASI300S.
-T300S-12	output is to a DASI300S in 12-pitch mode.
-T4014	output is to a Tektronix 4014.
-T37	output is to a TELETYPE Model 37.
-T382	output is to a DTC-382.
-T4000A	output is to a Trendata 4000A.
-TX	output is prepared for an EBCDIC lineprinter.
-Thp	output is to a HP264x (implies -c).
-T43	output is to a TELETYPE Model 43 (implies -c).
-T40/4	output is to a TELETYPE Model 40/4 (implies -c).
-T745	output is to a Texas Instrument 700 series terminal (implies -c).
-T2631	output is prepared for a HP2631 printer (where -T2631-e and -T2631-c may be used for expanded and compressed modes, respectively) (implies -c).
-Tlp	output is to a device with no reverse or partial line motions or other special features (implies -c).

Any other -T option given does not produce an error; it is equivalent to -Tlp.

A similar command is available for use with **troff** (see **mmt**[1]).

5.5.2 Using the -cm or -mm Flag

The **-mm** package can also be invoked by including the **-cm** or **-mm** flag on a **troff** or **nroff** command line. The **-cm** flag causes the compacted version of the macros to be loaded. The **-mm** flag causes the file `/usr/lib/tmac/tmac.m` to be read and processed before any other files. This action defines the **-mm** macros, sets default values for various parameters, and initializes the formatter to be ready to process the files of input text.

5.5.3 Typical Command Lines

The following prototype command lines are used with the various options explained in Chapters 1 and 2 of this section.

- Text without tables or equations:

```

mm [options] filename ...
or nroff [options] -cm filename ...
mmt [options] filename ...
or troff [options] -cm filename ...

```

- Text with tables:

```

mm -t [options] filename ...
or tbl filename ... | nroff [options] -cm
mmt -t [options] filename ...
or tbl filename ... | troff [options] -cm

```

- Text with equations:

```

mm -e [options] filename ...
or neqn filename ... | nroff [options] -cm
mmt -e [options] filename ...
or eqn filename ... | troff [options] -cm

```

- Text with both tables and equations:

```

mm -t -e [options] filename ...
or tbl filename ... | neqn | nroff [options] -cm
mmt -t -e [options] filename ...
or tbl filename ... | eqn | troff [options] -cm

```

When formatting a document with **nroff**, the file should normally be processed for a known terminal, since output may require terminal-specific features (e.g., reverse paper motion or half-line paper motion in both directions). Some commonly available terminal types and the command lines appropriate for them are given below.

- DASI450 in 10-pitch, 6 lines/inch mode, with .75 inch offset, and a line length of 6.0 inches (60 characters) where this is the default terminal type so no **-T** option is needed (unless **\$TERM** is set to another value):

```

mm filename ...
or nroff -T450 -h -cm filename ...

```

- DASI450 in 12-pitch, 6 lines/inch mode, with inches (72 characters):

```

mm -12 filename ...
or nroff -T450-12 -h -cm filename ...

```

or, to increase the line length to 80 characters and decrease the offset to 3 characters:

```

mm -12 -rW80 -rO3 filename ...
or nroff -T450-12 -rW80 -rO3 -h -cm filename ...

```

- Hewlett-Packard HP264x CRT family:

mm -T_{hp} filename ...
or nroff -cm filename ... | **col** | **hp**

- Any terminal incapable of reverse paper motion and also lacking hardware tab stops (Texas Instruments 700 series, etc.):

mm -T₇₄₅ filename ...
or nroff -cm filename ... | **col -x**

- Versatec raster plotter:

vp [vp-options] "**mm -rT₂ -c** filename ..."
or vp [vp-options] "**nroff -rT₂ -cm** filename ... | **col**"

Of course, if **tbl**[1] and **eqn**[1]/**neqn**[1] are needed, they must be invoked as shown in the command line prototypes at the beginning of this section.

If two-column processing is used with **nroff**, either the **-c** option must be specified to **nroff** or the output must be postprocessed by **col**[1]. In the latter case, the **-T₃₇** terminal type must be specified to **nroff**, the **-h** option must *not* be specified, and the output of **col**(1) must be processed by the appropriate terminal filter (e.g., **450**(1)); **mm**(1) with the **-c** option handles all this automatically.

Note: **Mm**(1) uses **col**(1) automatically for many of the terminal types.

5.5.4 Parameters Set on the Command Line

Mm uses various number registers to hold parameter values that govern certain aspects of output style. Many of these registers can be changed within the text files via **.nr** requests. In addition, some of these registers can be set from the command line itself, a useful feature for those parameters that should not be permanently embedded within the input text itself. If used, these registers (with the possible exception of the register **P**—see below) must be set on the command line (or before the **-mm** macro definitions are processed) and their meanings are:

- rAn for **n** = 1 has the effect of invoking the **.AF** macro without an argument.
- rCn **n** sets the type of copy (e.g., DRAFT) to be printed at the bottom of each page.
 - n** = 3 for DRAFT with single-spacing and default paragraph style.
 - n** = 4 for DRAFT with double-spacing and 10 space paragraph indent.
- rD1 sets debug mode. This flag requests the formatter to attempt to continue processing even if **-mm** detects errors that would otherwise cause termination. It also includes

some debugging information in the default page header.

- rEn** controls the font of the Subject/Date/From fields. If *n* is 0, then these fields are bold (default for **troff**), and if *n* is 1, then these fields are regular text (default for **nroff**).
- rLk** sets the length of the physical page to *k* lines.

Note: For **nroff**, *k* is an *unscaled* number representing lines or character positions; for **troff**, *k* must be *scaled*.

The default value is 66 lines per page. This parameter is used, for example, when directing output to a Versatec printer.

- rNn** specifies the page numbering style. When *n* is 0 (default), all pages get the (prevailing) header. When *n* is 1, the page header replaces the footer on page 1 only. When *n* is 2, the page header is omitted from page 1. When *n* is 3, “section-page” numbering for footnote and reference numbering in sections occurs). When *n* is 4, the *default* page header is suppressed; however a user-specified header is not affected. When *n* is 5, “section-page” and “section-figure” numbering occurs.

<i>n</i>	<i>Page 1</i>	<i>Pages 2 ff.</i>
0	header	header
1	header replaces footer	header
2	no header	header
3	“section-page” as <i>footer</i>	
4	no header	no header unless PH defined
5	same as 3—with “section-figure”	

The contents of the prevailing header and footer do not depend on the value of the number register *N*; *N* only controls whether and where the header (and, for *N*=3 or 5, the footer) is printed, as well as the page numbering style. In particular, if the header and footer are null, the value of *N* is irrelevant.

- rOk** offsets output *k* spaces to the right.

Note: For **nroff**, these values are unscaled numbers representing lines or character positions. For **troff**, these values must be scaled.

It is helpful for adjusting output positioning on some terminals. The default offset if this register is not set on the command line is .75 inches.

Note: The register name is the capital letter “O”, not the digit zero (0).

- rP*n* specifies that the pages of the document are to be numbered starting with *n*. This register may also be set via a .nr request in the input text.
- rS*n* sets the point size and vertical spacing for the document. The default *n* is 10, i.e., 10-point type on 12-point leading (vertical spacing), giving 6 lines per inch. This parameter applies to **troff** only.
- rT*n* provides register settings for certain devices. If *n* is 1, then the line length and page offset are set to 80 and 3, respectively. Setting *n* to 2 changes the page length to 84 lines per page and inhibits underlining; it is meant for output sent to the Versatec raster plotter. The default value for *n* is 0. This parameter applies to **nroff** only.
- rU1 controls underlining of section headings. This flag causes only letters and digits to be underlined. Otherwise, all characters (including spaces) are underlined. This parameter applies to **nroff** only.
- rW*k* page width (i.e., line length and title length) is set to *k*. This can be used to change the page width from the default value of 6.0 inches (60 characters in 10 pitch or 72 characters in 12 pitch).

5.5.5 Omission of -cm or -mm

If a large number of arguments is required on the command line, it may be convenient to set up the first (or only) input file of a document as follows:

```
zero or more initializations of registers
listed above
.so /usr/lib/tmac/tmac.m
remainder of text
```

If you choose this route, do not use either the -cm or -mm command line flag or the mm/mmt commands. The .so request has the equivalent effect, but the registers must be initialized before the .so request. Values in these registers are meaningful only if set before the macro definitions are processed. This method is used to “lock” into the input file those parameters that are seldom changed. For example;

```
.nr W 80
.nr O 10
.nr N 3
.so /usr/lib/tmac/tmac.m
.H 1 "INTRODUCTION"
:
```

specifies, for **nroff**, a line length of 80, a page offset of 10, and "section-page" numbering.

5.6 FORMATTING CONCEPTS

5.6.1 Basic Terms

Both **troff** and **nroff** normally fill output lines with text from input lines. The output lines may be justified so that both the left and right margins are aligned. As the lines are being filled, words may also be hyphenated, as necessary. It is possible to turn any of these modes on and off. Turning off fill mode also turns off justification and hyphenation.

Certain formatting commands (requests and macros) cause the filling of the current output line to cease, the line (of whatever length) to be printed, and the subsequent text to begin a new output line. This printing of a partially filled output line is known as a break. A few formatter requests and most of the **-mm** macros cause a break.

While formatter requests can be used with **-mm**, you need to understand the consequences and side-effects that each such request might have. Actually, there is little need to use formatter requests. The **-mm** macros handle a variety of formats, and can be used in all but the most specialized cases. We suggest that you use formatter requests only when absolutely necessary.

Note: Each new sentence must begin on a new line.

5.6.2 Arguments and Double Quotes

For any macro call, a null argument is an argument whose width is zero. Such an argument often has a special meaning; the preferred form for a null argument is "".

Note: Omitting an argument is not the same as supplying a null argument.

Furthermore, omitted arguments can occur only at the end of an argument list, while null arguments can occur anywhere.

Any macro argument containing ordinary (paddable) spaces must be enclosed in double quotes (""). Otherwise, it will be treated as

several separate arguments.

Note: The double quote (") is a single character that must not be confused with two apostrophes or acute accents (' '), or with two grave accents (` `).

Double quotes (") are not permitted as part of the value of a macro argument or of a string that is to be used as a macro argument. If you must, use two grave accents (` `) and/or two acute accents (' ') instead. This restriction is necessary because macro arguments may be processed (interpreted) an arbitrary number of times. For example, headings that are first printed in the text may be (re)printed in the table of contents.

5.6.3 Unpaddable Spaces

When output lines are justified, existing spaces in a line may have additional spaces appended to them. This may harm the desired alignment of text. To avoid this problem, it is necessary to be able to specify a space that cannot be expanded during justification, — an “unpaddable” space. There are several ways to accomplish this.

A common mechanism for making a space unpaddable is to precede the space with a backslash (“\ ”). Another way is to specify that some (seldom used) character is to be translated into a space upon output. Because this translation occurs after justification, the chosen character may be used anywhere an unpaddable space is desired. The tilde (~) is often used for this purpose. To use it in this way, insert the following at the beginning of the document:

```
.tr ~
```

If a tilde must actually appear in the output, it can be temporarily “recovered” by inserting:

```
.tr ~ ~
```

before the place where it is needed. Its previous usage is restored by repeating the “.tr ~ ”, but only after a break or after the line containing the tilde has been forced out. Note that the use of the tilde in this fashion is not recommended for documents in which the tilde is used within equations.

5.6.4 Hyphenation

The formatters do not perform hyphenation unless you request it. Hyphenation can be turned on in the body of the text by specifying:

```
.nr Hy 1
```

exactly once at the beginning of the document.

If hyphenation is requested, the formatter will automatically hyphenate words where it can. You may specify explicit

hyphenation points for a specific occurrence of any word by the use of a special character known as a hyphenation indicator. You may also specify hyphenation points for a small (about 128 characters) list of words.

If the hyphenation indicator (initially, the two-character sequence “\%”) appears at the beginning of a word, the word is not hyphenated. Alternatively, the hyphenation character can be used to indicate preferred hyphenation point(s) inside a word. In any case, all occurrences of the hyphenation indicator disappear on output.

You may specify a different hyphenation indicator:

.HC [hyphenation-indicator]

The circumflex () is often used for this purpose. To do this, insert the following at the beginning of a document:

.HC

Note: The formatter is always allowed to break a word after a hyphen or a dash (em dash), and will do so if necessary whether or not hyphenation is on.

You may supply, via the .hw request, a small list of words with the proper hyphenation points indicated. For example, to indicate the proper hyphenation of the word “printout,” one may specify:

.hw print-out

5.6.5 Tabs

The MT and CS macros use the formatter’s .ta request to set tab stops, and then restore the default values. Default tabs are every eight characters in **nroff**; every inch in **troff**. You must explicitly reset tabs if you need other values. A tab character is always interpreted with respect to its position on the input line, rather than its position on the output line. In general, tab characters should appear only on lines processed in “no-fill” mode.

Also, note that **tbl**[1] changes tab stops, but does not restore the default tab settings.

5.6.6 Special Use of the BEL Character

The non-printing character BEL (\$07) is used as a delimiter in many macros where it is necessary to compute the width of an argument or to delimit arbitrary text, e.g., in headers and footers, and list marks. That’s why BEL characters should not appear in your input text (especially in arguments to macros).

5.6.7 Bullets

A bullet (•) is often obtained on a typewriter terminal by using an “o” overstruck by a “+”. For compatibility with **troff**, a bullet string is provided by **-mm**. Rather than overstriking, use the sequence:

```
\*(BU
```

wherever a bullet is desired. Note that the bulleted list (.BL) macros use this string to automatically generate the bullets for the list items.

5.6.8 Dashes, Minus Signs, and Hyphens

Troff has distinct characters for a dash, a minus sign, and a hyphen, while **nroff** does not. Those who intend to use **nroff** only may use the minus sign (“-”) for all three.

Those who wish mainly to use **troff** should follow the **troff** escape conventions described in Chapters 1 and 2 of this section.

Those who want to use both formatters must take care during text preparation. Unfortunately, these characters cannot be represented in a way that is both compatible and convenient. We suggest the following approach:

- | | |
|--------|--|
| Dash | Type <code>*(EM</code> for each text dash for both nroff and troff . This string generates an em dash (—) in troff and generates “-” in nroff . Note that the dash list (.DL) macros automatically generate the em dashes for the list items. |
| Hyphen | Type “-” and use as is for both formatters. Nroff will print it as is, and troff will print a true hyphen. |
| Minus | Type “\”-” for a true minus sign, regardless of formatter. Nroff will effectively ignore the “\”, while troff will print a true minus sign. |

5.6.9 Trademark String

This string (`*(Tm`) places the letters “TM” one-half line above the text that it follows.

For example,

```
The UNIX\*(Tm Operating System is now widely-supported
and running on many types of computing machinery.
```

yields:

```
The UNIXTM Operating System is now widely-supported
and running on many types of computing machinery.
```

5.7 PARAGRAPHS AND HEADINGS

This section describes simple paragraphs and section headings. Additional paragraph and list styles are covered in later sections.

5.7.1 Paragraphs

`.P [type]`
one or more lines of text.

This macro is used to begin two kinds of paragraphs. In a left-justified paragraph, the first line begins at the left margin, while in an indented paragraph, the first line is indented five spaces.

A document's default paragraph style is obtained by specifying `“.P”` before each paragraph that does not follow a heading. The default style is controlled by the register *Pt*. The initial value of *Pt* is 0, which always provides left-justified paragraphs. All paragraphs can be force-indented by inserting the following at the beginning of the document:

```
.nr Pt 1
```

All paragraphs will be indented except after headings, lists, and displays if the following:

```
.nr Pt 2
```

is inserted at the beginning of the document.

The amount a paragraph is indented is contained in the register *Pi*, whose default value is 5. To indent paragraphs by, say, 10 spaces, insert:

```
.nr Pi 10
```

at the beginning of the document. Of course, both the *Pi* and *Pt* register values must be greater than zero for any paragraphs to be indented.

The number register *Ps* controls the amount of spacing between paragraphs. By default, *Ps* is set to 1, yielding one blank space (a vertical space).

Note: Values that specify indentation must be unscaled and are treated as “character positions,” i.e., as a number of ens. In **troff**, an en is the number of points (1 point = 1/72 of an inch) equal to half the current point size. In **nroff**, an en is equal to the width of a (fixed-width) character.

Regardless of the value of *Pt*, an individual paragraph can be forced to be left-justified or indented. `“.P 0”` always forces left justification; `“.P 1”` always causes indentation by the amount specified by the register *Pi*.

If `.P` occurs inside a list, the indent (if any) of the paragraph is added to the current list indent.

Numbered paragraphs may be produced by setting the register `Np` to 1. This produces paragraphs numbered within first level headings, e.g., 1.01, 1.02, 1.03, 2.01, etc.

A different style of numbered paragraphs is obtained by using the

```
.nP
```

macro rather than the `.P` macro for paragraphs. This produces paragraphs that are numbered within second level headings and contain a “double-line indent” in which the text of the second line is indented to be aligned with the text of the first line so that the number stands out.

```
.H 1 "FIRST HEADING"
```

```
.H 2 "Second Heading"
```

```
.nP
```

```
one or more lines of text
```

5.7.2 Numbered Headings

```
.H level [heading-text] [heading-suffix]
```

```
zero or more lines of text
```

The `.H` macro provides seven levels of numbered headings, as illustrated by this document. Level 1 is the highest (most major); level 7 the lowest.

The heading-suffix is appended to the heading-text and may be used for footnote marks which should not appear with the heading text in the Table of Contents.

Note: The `.H` macro also performs the function of the `.P` macro. If a `.P` follows a `.H`, the `.P` is ignored.

The effect of `.H` varies according to the *level* argument. First-level headings are preceded by two blank lines (one vertical space); all others are *preceded* by one blank line (one-half of a vertical space).

`.H 1 heading-text` gives a bold heading *followed* by a single blank line (one-half of a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Full capital letters should normally be used to make the heading stand out.

`.H 2 heading-text` yields a bold heading followed by a single blank line (one-half of a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Normally, initial capitals are used.

`.H n heading-text` for $3 \leq n \leq 7$, produces an underlined (Italic) heading followed by two spaces. The following text appears on the same line, i.e., these are run-in headings.

Appropriate numbering and spacing (horizontal and vertical) occur even if the heading text is omitted from a `.H` macro call.

5.7.3 Altering Heading Pre-Space

A first-level heading normally has two blank lines (one vertical space) preceding it, and all others have one blank line (one-half of a vertical space). A multi-line heading that would split across a page break is, instead, automatically moved to the top of the next page. Every first-level heading may be forced to the top of a new page by inserting:

```
.nr Ej 1
```

at the beginning of the document. Long documents may be made more manageable if each section starts on a new page. Setting `Ej` to a higher value causes the same effect for headings up to that level, i.e., a page eject occurs if the heading level is less than or equal to `Ej`.

5.7.4 Heading Post-Space

Three registers control the appearance of text immediately following a `.H` call. They are `Hb` (heading break level), `Hs` (heading space level), and `Hi` (post-heading indent).

If the heading level is less than or equal to `Hb`, a break occurs after the heading. If the heading level is less than or equal to `Hs`, a blank line (one-half of a vertical space) is inserted after the heading. Defaults for `Hb` and `Hs` are 2. If a heading level is greater than `Hb` and also greater than `Hs`, then the heading (if any) is run into the following text. These registers permit headings to be separated from the text in a consistent way throughout a document, while allowing easy alteration of white space and heading emphasis.

For any stand-alone heading, i.e., a heading not run into the following text, the alignment of the next line of output is controlled by the register `Hi`. If `Hi` is 0, text is left-justified. If `Hi` is 1 (the default value), the text is indented according to the paragraph type as specified by the register `Pt`. Finally, if `Hi` is 2, text is indented to line up with the first word of the heading itself, so that the heading number stands out more clearly.

For example, to cause a blank line (one-half of a vertical space) to appear after the first three heading levels, to have no run-in headings, and to force the text following all headings to be left-justified (regardless of the value of `Pt`), the following should appear at the top of the document:

```
.nr Hs 3
.nr Hb 7
.nr Hi 0
```

5.7.5 Centered Headings

The register *Hc* can be used to obtain centered headings. A heading is centered if its level is less than or equal to *Hc*, and if it is also stand-alone. *Hc* is 0 initially (no centered headings).

5.7.6 Format Control by Heading Level

Any heading that is underlined by **nroff** is made Italic by **troff**. The string *HF* (heading font) contains seven codes that specify the fonts for heading levels 1-7. The legal codes, their interpretations, and the defaults for *HF* are:

<i>Formatter</i>	<i>HF Code</i>			<i>Default HF</i>
	<i>1</i>	<i>2</i>	<i>3</i>	
nroff	no underline	underline	bold	3 3 2 2 2 2 2
troff	Roman	Italic	bold	3 3 2 2 2 2 2

Thus, levels 1 and 2 are bold; levels 3 through 7 are underlined in **nroff** and Italic in **troff**. You may reset *HF* as desired. Any value omitted from the rightmost side of the list is taken to be 1. For example, the following would result in five bold levels and two non-underlined (roman) levels:

```
.ds HF 3 3 3 3 3
```

5.7.7 Nroff Heading Underlining Styles

Nroff can underline in two ways. The normal style (*.ul* request) is to underline only letters and digits. The continuous style (*.cu* request) underlines all characters, including spaces. By default, **-mm** attempts to use the continuous style on any heading that is to be underlined and is short enough to fit on a single line. If a heading is to be underlined, but is too long, it is underlined the normal way (i.e., only letters and digits are underlined).

All underlining of headings can be forced to the normal way by using the *-rU1* flag when invoking **nroff**.

5.7.8 Heading Point Sizes

You may also specify the desired point size for each heading level with the *HP* string (for use with **troff** only).

```
.ds HP [ps1] [ps2] [ps3] [ps4] [ps5] [ps6] [ps7]
```

By default, the text of headings (*.H* and *.HU*) is printed in the same point size as the body except that bold stand-alone headings are printed in a size one point smaller than the body. The string *HP*, similar to the string *HF*, can be specified to contain up to seven values, corresponding to the seven levels of headings. For example:

```
.ds HP 12 12 10 10 10 10 10
```

specifies that the first and second level headings are to be printed in 12-point type, with the remainder printed in 10-point. Note that the specified values may also be relative point-size changes, e.g.:

```
.ds HP +2 +2 -1 -1
```

If absolute point sizes are specified, then those sizes will be used regardless of the point size of the body of the document. If relative point sizes are specified, then the point sizes for the headings will be relative to the point size of the body, even if the latter is changed.

Omitted or zero values imply that the default point size will be used for the corresponding heading level.

Note: Only the point size of the headings is affected. Specifying a large point size without providing increased vertical spacing (via `.HX` and/or `.HZ`) may cause overprinting.

5.7.9 Marking Styles

```
.HM [arg1] ... [arg7]
```

The registers named *H1* through *H7* are used as counters for the seven levels of headings. Their values are normally printed using Arabic numerals. The `.HM` macro (heading mark style) allows this choice to be overridden, thus providing “outline” and other document styles. This macro can have up to seven arguments; each argument is a string indicating the type of marking to be used. Legal values and their meanings are shown below; omitted values are interpreted as 1, while illegal values have no effect.

<i>Value</i>	<i>Interpretation</i>
1	Arabic (default for all levels)
0001	Arabic with enough leading zeroes to get the specified number of digits
A	Uppercase alphabetic
a	Lowercase alphabetic
I	Uppercase Roman
i	Lowercase Roman

By default, the complete heading mark for a given level is built by concatenating the mark for that level to the right of all marks for all levels of higher value. To inhibit the concatenation of heading level marks, i.e., to obtain just the current-level mark followed by a period, set the register *Ht* (heading-mark type) to 1.

For example, a commonly-used “outline” style is obtained by:

```
.HM I A 1 a i
.nr Ht 1
```

5.7.10 Unnumbered Headings

```
.HU heading-text
```

`.HU` is a special case of `.H`; it is handled in the same way as `.H`, except that no heading mark is printed. In order to preserve the hierarchical structure of headings when `.H` and `.HU` calls are intermixed, each `.HU` heading is considered to exist at the level given by register *Hu*, whose initial value is 2. Thus, in the normal case, the only difference between:

```
.HU heading-text
```

and

```
.H 2 heading-text
```

is the printing of the heading mark for the latter. Both have the effect of incrementing the numbering counter for level 2, and resetting to zero the counters for levels 3 through 7. Typically, the value of *Hu* should be set to make unnumbered headings (if any) be the lowest-level headings in a document.

`.HU` can be especially helpful in setting up Appendices and other sections that may not fit well into the numbering scheme of the main body of a document.

5.7.11 Headings and the Table of Contents

The text of headings and their corresponding page numbers can be automatically collected for a table of contents. This is accomplished by doing the following two things:

- specifying in the register *Cl* what level headings are to be saved
- invoking the `.TC` macro at the end of the document

Any heading whose level is less than or equal to the value of the register *Cl* (contents level) is saved and later displayed in the table of contents. The default value for *Cl* is 2, i.e., the first two levels of headings are saved.

Due to the way the headings are saved, it is possible to exceed the formatter's storage capacity, particularly when saving many levels of many headings while also processing displays. If this happens, the "Out of temp file space" diagnostic will be issued; the only remedy is to save fewer levels and/or to have fewer words in the heading text.

5.7.12 Page Numbering Style

By default, pages are numbered sequentially at the top of the page. For large documents, you may want to use page numbering of the form *section*-page, where *section* is the number of the current

first-level heading. This page numbering style can be achieved by specifying the `-rN3` or `-rN5` flag on the command line. This also has the effect of setting *Ej* to 1, i.e., each section begins on a new page. In this style, the page number is printed at the bottom of the page, so that the correct section number is printed.

5.7.13 User Exit Macros

Note: This section is intended only for those who are accustomed to writing formatter macros.

```
.HX dlevel rlevel heading-text
.HY dlevel rlevel heading-text
.HZ dlevel rlevel heading-text
```

The `.HX`, `.HY`, and `.HZ` macros allow you to obtain a final level of control over the previously described heading mechanism. `-mm` calls but does not define `.HX`, `.HY`, and `.HZ`. They are available for use as needed in preparing custom heading treatments. The `.H` macro invokes `.HX` shortly before the actual heading text is printed; it calls `.HZ` as its last action. After `.HX` is invoked, the size of the heading is calculated. This processing causes certain features that may have been included in `.HX`, such as `.ti`, for temporary indent, to be lost. After the size calculation, `.HY` is invoked so that the user may respecify these features. All the default actions occur if these macros are not defined. If the `.HX`, `.HY`, or `.HZ` are defined by the user, the user-supplied definition is interpreted at the appropriate point. These macros can therefore influence the handling of all headings, because the `.HU` macro is actually a special case of the `.H` macro.

If you originally invoked the `.H` macro, then the derived level (*dlevel*) and the real level (*rlevel*) are both equal to the level given in the `.H` invocation. If you originally invoked the `.HU` macro, *dlevel* is equal to the contents of register *Hu*, and *rlevel* is 0. In both cases, *heading-text* is the text of the original invocation.

By the time `.H` calls `.HX`, it has already incremented the heading counter of the specified level, produced blank line(s) (vertical space) to precede the heading, and accumulated the "heading mark", i.e., the string of digits, letters, and periods needed for a numbered heading. When `.HX` is called, all user-accessible registers and strings can be referenced, as well as the following:

```
string }0      If rlevel is non-zero, this string contains the
                "heading mark." Two unpaddable spaces (to
                separate the mark from the heading ) have been
                appended to this string. If rlevel is 0, this string is
                null.
```

- register ;0 This register indicates the type of spacing that is to follow the heading. A value of 0 means that the heading is run-in. A value of 1 means a break (but no blank line) is to follow the heading. A value of 2 means that a blank line (one-half of a vertical space) is to follow the heading.
- string }2 If register ;0 is 0, this string contains two unpaddable spaces that will be used to separate the (run-in) *heading* from the following *text*. If register ;0 is non-zero, this string is null.
- register ;3 This register contains an adjustment factor for a .ne request issued before the heading is actually printed. On entry to .HX, it has the value 3 if *dlevel* equals 1, and 1 otherwise. The .ne request is for the following number of lines: the contents of the register ;0 taken as blank lines (halves of vertical space) plus the contents of register ;3 as blank lines (halves of vertical space) plus the number of lines of the heading.

You may alter the values of }0, }2, and ;3 within .HX as desired. The following are examples of actions that might be performed by defining .HX to include the lines shown:

Change first-level heading mark from format *n.* to *n.0*:
`.if \\$1=1 .ds }0 \\n(H1.0\ \ (stands for a space)`

Separate run-in heading from the text with a period and two unpaddable spaces:

`.if \\n(;0=0 .ds }2 .\ \`

Assure that at least 15 lines are left on the page before printing a first-level heading:

`.if \\$1=1 .nr ;3 15-\\n(;0`

Add 3 additional blank lines before each first-level heading:

`.if \\$1=1 .sp 3`

Indent level 3 run-in headings by 5 spaces:

`.if \\$1=3 .ti 5n`

If temporary string or macro names are used within .HX, care must be taken in the choice of their names.

.HY is called after the .ne is issued. Certain features requested in .HX must be repeated. For example,

```
.de HY
.if \\$1=3 .ti 5n
..
```

.HZ is called at the end of .H to permit user-controlled actions after the heading is produced. For example, in a large document, sections may correspond to chapters of a book, and you may want to change a page header or footer. For example,

```
.de HZ
.if \\$1=1 .PF " ' ' Section \\$3 ' ' "
..
```

5.7.14 Hints for Large Documents

A large document is often organized into one file per section. If you do this, use enough digits in the names of these files to accommodate the maximum number of sections, i.e., use suffix numbers 01 through 20 rather than 1 through 9 and 10 through 20.

If, when formatting individual sections of long documents, you need to retain the correct section numbers, set register *H1* to 1 less than the number of the section just before the corresponding “.H 1” call. For example, at the beginning of section 5, insert:

```
.nr H1 4
```

Note: This practice defeats the automatic (re)numbering of sections when sections are added or deleted. Be sure to remove such lines when they are no longer needed.

5.8 LISTS

This section explains how you can use **-mm** macros to obtain many different kinds of lists: automatically-numbered and alphabetized lists, bulleted lists, dash lists, lists with arbitrary marks, and lists starting with arbitrary strings.

5.8.1 The Parts of a List

All lists are composed of the following parts:

- A list-initialization macro that controls such things as line spacing, indentation, marking with special symbols, and numbering or alphabetizing.
- One or more List Item (.LI) macros, each followed by the actual text of the corresponding list item.
- The List End (.LE) macro that terminates the list and restores the margin(s).

Lists may be nested up to five levels deep. The list-initialization macro saves the previous list status (indentation, marking style, etc.); the .LE macro restores it.

With this approach, the format of a list is specified only at the beginning of that list. In addition, by building on the existing

structure, you may create your own customized sets of list macros.

5.8.2 Sample Nested Lists

The input for several lists and the corresponding output are shown below. The `.AL` and `.DL` macro calls contained therein are examples of list-initialization macros. This example will help us to explain the material in the following sections. Input text:

```
.AL A
.LI
This is an alphabetized item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.AL
.LI
This is a numbered item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.DL
.LI
This is a dash item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LI + 1
This is a dash item with a ``plus'' as prefix.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LE
.LI
This is numbered item 2.
.LE
.LI
This is another alphabetized item, B.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LE
.P
This paragraph appears at the left margin.
```


Output:

- A. This is an alphabetized item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.
1. This is a numbered item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.
 - This is a dash item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.
 - + — This is a dash item with a "plus" as prefix. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.
 2. This is numbered item 2.
- B. This is another alphabetized item, B. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

This paragraph appears at the left margin.

5.8.3 Common List Macros

Because all lists share the same overall structure except for the list-initialization macro, we first discuss the macros common to all lists.

5.8.3.1 List Item

```
.LI [mark] [1]
one or more lines of text that make up the list item.
```

The `.LI` macro is used with all lists. It normally causes the output of a single blank line (one-half of a vertical space) before its item, although this may be suppressed. If no arguments are given, it labels its item with the current mark, which is specified by the most recent list-initialization macro. If a single argument is given to `.LI`, that argument is output instead of the current mark. If two arguments are given, the first argument becomes a prefix to the current mark, thus allowing the user to emphasize one or more items in a list. One unpaddingable space is inserted between the prefix and the mark. For example:

```
.BL 6
.LI
This is a simple bullet item.
.LI +
This replaces the bullet with a ``plus.``
.LI + 1
But this uses ``plus`` as prefix to the bullet.
.LE
```

yields:

- This is a simple bullet item.
- + This replaces the bullet with a “plus.”
- + • But this uses “plus” as prefix to the bullet.

Note: The *mark* must not contain ordinary (paddable) spaces. Alignment of items will be lost if the right margin is justified

If the *current mark* (in the *current list*) is a null string, and the first argument of `.LI` is omitted or null, the first line of the following text will be “outdented,” starting at the place where the *mark* would have started. This format is commonly known as a “hanging” indent.

5.8.3.2 List End

```
.LE [1]
```

The List End macro restores the state of the list to that which existed prior to the most recent list-initialization macro call. If the optional argument is given, `.LE` outputs a blank line (one-half of a vertical space). This option should generally be used only when the `.LE` is followed by running text, but not when followed by a macro that produces blank lines of its own, such as `.P`, `.H`, or `.LI`.

`.H` and `.HU` automatically clear all list information, so you may legally omit the `.LE(s)` that would normally occur just before either of these macros, although this practice is not recommended. (Errors will occur if the list text is separated from the heading at some later time e.g., by insertion of text.)

5.8.4 List Initialization Macros

These macros are actually implemented as calls to the more primitive `.LB` macro.

5.8.4.1 Numbered and Alphabetized Lists

```
.AL [type] [text-indent] [1]
```

The `.AL` macro is used to begin sequentially-numbered or alphabetized lists. If there are no arguments, the list is numbered, and text

is indented *Li*, initially 6 *ens*.

Note: Values that specify indentation must be unscaled and are treated as “character positions,” i.e., as a number of *ens* from the indent in force when the *.AL* is called, thus leaving room for a space, two digits, a period, and two spaces before the text.

Spacing at the beginning of the list and between the items can be suppressed by setting the *Ls* (list space) register. *Ls* is set to the innermost list level for which spacing *is* done. For example,

```
.nr Ls 0
```

specifies that no spacing will occur around any list items. The default value for *Ls* is 6 (which is the *maximum* list nesting level).

The *type* argument may be given to obtain a different type of sequencing, and its value should indicate the first element in the sequence desired, i.e., it must be 1, A, a, I, or i. If *type* is omitted or null, then “1” is assumed. If *text-indent* is non-null, it is used as the number of spaces from the current indent to the text, i.e., it is used instead of *Li* for this list only. If *text-indent* is null, then the value of *Li* will be used.

If the third argument is given, a blank line (one-half of a vertical space) will *not* separate the items in the list. A blank line (one-half of a vertical space) will occur before the first item, however.

5.8.4.2 Bulleted List

```
.BL [text-indent] [1]
```

.BL begins a bulleted list, in which each item is marked by a bullet (•) followed by one space. If *text-indent* is non-null, it overrides the default indentation—the amount of paragraph indentation as given in the register *Pi*. In the default case, the text of bullet and dash lists lines up with the first line of indented paragraphs.

If a second argument is specified, no blank lines will separate the items in the list.

5.8.4.3 Dashed List

```
.DL [text-indent] [1]
```

.DL is identical to *.BL*, except that a dash is used instead of a bullet.

5.8.4.4 Marked List

```
.ML mark [text-indent] [1]
```

.ML is much like *.BL* and *.DL*, but expects the user to specify an arbitrary mark, which may consist of more than a single character.

Text is indented *text-indent* spaces if the second argument is not null; otherwise, the text is indented one more space than the width of *mark*. If the third argument is specified, no blank lines will separate the items in the list.

Note: The *mark* must not contain ordinary (paddable) spaces. Alignment of items will be lost if the right margin is justified.

5.8.4.5 Reference List

`.RL [text-indent] [1]`

A `.RL` call begins an automatically-numbered list in which the numbers are enclosed by square brackets ([]). *Text-indent* may be supplied, as for `.AL`. If omitted or null, it is assumed to be 6, a convenient value for lists numbered up to 99. If the second argument is specified, no blank lines will separate the items in the list.

5.8.4.6 Variable-Item List

`.VL text-indent [mark-indent] [1]`

When a list begins with a `.VL`, there is effectively no *current mark*. It is expected that each `.LI` will provide its own mark. This form is typically used to display definitions of terms or phrases. *Mark-indent* gives the number of spaces from the current indent to the beginning of the *mark*. It defaults to 0 if omitted or null. *Text-indent* gives the distance from the current indent to the beginning of the text. If the third argument is specified, no blank lines will separate the items in the list. Here is an example of `.VL` usage:

```
.tr ~
.VL 20 2
.LI mark~ 1
Here is a description of mark 1;
``mark 1`` of the .LI line contains a tilde translated
to an unpaddable space in order to avoid extra spaces between
``mark`` and ``1``.
.LI second~ mark
This is the second mark, also using a tilde translated to an unpaddable space.
.LI third~ mark~ longer~ than~ indent:
This item shows the effect of a long mark; one space separates the mark
from the text.
.LI ~
This item effectively has no mark because the
tilde following the .LI is translated into a space.
.LE
```

yields:

mark 1 Here is a description of mark 1; “mark 1” of the .LI line contains a tilde translated to an unpaddable space in order to avoid extra spaces between “mark” and “1”.

second mark This is the second mark, also using a tilde translated to an unpaddable space.

third mark longer than indent:

This item shows the effect of a long mark; one space separates the mark from the text.

This item effectively has no mark because the tilde following the .LI is translated into a space.

The tilde argument on the last .LI above is required; otherwise a hanging indent would have been produced. A hanging indent is produced by using .VL and calling .LI with no arguments or with a null first argument. For example:

```
.VL 10
.LI
Here is some text that shows a hanging indent.
The first line of text is at the left margin.
The second is indented 10 spaces.
.LE
```

yields:

Here is some text that shows a hanging indent. The first line of text is at the left margin. The second is indented 10 spaces.

Note: The *mark* must not contain ordinary (paddable) spaces. Alignment of items will be lost if the right margin is justified

5.8.5 List-Begin Macro and Customized Lists

Note: This section is intended only for those who are accustomed to writing formatter macros.

```
.LB text-indent mark-indent pad type [mark] [LI-space] [LB-space]
```

The list-initialization macros described above suffice for almost all cases. However, experienced macro hands may obtain more control over the layout of lists by using the primitive list-begin macro .LB, the starting point for all the other list-initialization macros. Its arguments are as follows:

Text-indent gives the number of spaces that the text is to be indented from the current indent. Normally, this value is taken from the register *Li* for automatic lists and from the register *Pi* for bulleted and dashed lists.

The combination of *mark-indent* and *pad* determines the placement of the mark. The mark is placed within an area (called *mark area*) that starts *mark-indent* spaces to the right of the current indent, and ends where the text begins (i.e., ends *text-indent* spaces to the right of the current indent).

Note: The *mark-indent* argument is typically 0.

Within the mark area, the mark is left-justified if *pad* is 0. If *pad* is *n* (and *n* is greater than 0), *n* blanks are appended to the mark, and the *mark-indent* value is ignored. The resulting string immediately precedes the text. That is, the mark is effectively right-justified *pad* spaces immediately to the left of the text.

Type and *mark* interact to control the type of marking used. If *type* is 0, simple marking is performed using the mark character(s) found in the *mark* argument. If *type* is greater than 0, automatic numbering or alphabetizing is done, and *mark* is then interpreted as the first item in the sequence to be used for numbering or alphabetizing, i.e., it is chosen from the set (1, A, a, I, i). That is:

<i>Type</i>	<i>Mark</i>	<i>Result</i>
0	omitted	hanging indent
0	<i>string</i>	<i>string</i> is the mark
>0	omitted	arabic numbering
>0	one of: 1, A, a, I, i	automatic numbering or alphabetic sequencing

Each non-zero value of *type* from 1 to 6 selects a different way of displaying the items. The following table, where *x* is the generated number or letter, shows the output appearance for each value of *type*:

<i>Type</i>	<i>Appearance</i>
1	x.
2	x)
3	(x)
4	[x]
5	<x>
6	{x}

Note: The *mark* must not contain ordinary (paddable) spaces. Alignment of items will be lost if the right margin is justified.

LI-space gives the number of blank lines (halves of a vertical space) that should be output by each `.LI` macro in the list. If omitted, *LI-space* defaults to 1; the value 0 can be used to obtain compact lists. If *LI-space* is greater than 0, the `.LI` macro issues a `.ne` request for two lines just before printing the mark.

LB-space, the number of blank lines (one-half of a vertical space) to be output by `.LB` itself, defaults to 0 if omitted.

There are three reasonable combinations of *LI-space* and *LB-space*. The normal case is to set *LI-space* to 1 and *LB-space* to 0, yielding one blank line *before* each item in the list; such a list is usually terminated with a `“LE 1”` to end the list with a blank line. In the second case, for a more compact list, set *LI-space* to 0 and *LB-space* to 1, and, again, use `“LE 1”` at the end of the list. The result is a list with one blank line before and after it. If you set both *LI-space* and *LB-space* to 0, and use `“LE”` to end the list, a list with **no** blank lines will result.

5.8.6 User-Defined List Structures

If a large document requires complex list structures, it is useful to be able to define the appearance for each list level only once, instead of having to define it at the beginning of each list. This permits consistency of style in a large document. For example, a generalized list-initialization macro might be defined in such a way that what it does depends on the list-nesting level in effect at the time the macro is called. Suppose that levels 1 through 5 of lists are to have the following appearance:

A.

[1]

•

a)

+

The following code defines a macro (`.aL`) that always begins a new list and determines the type of list according to the current list level. To understand it, you should know that the number register `:g` is used by the `-mm` list macros to determine the current list level; it is 0 if there is no currently active list. Each call to a list-initialization macro increments `:g`, and each `.LE` call decrements it.

```

.de aL
  '\ register g is used as a local temporary
  '\ to save :g before it is changed below
  .nr g \n(:g
  .if \ng=0 .AL A \ give me an A.
  .if \ng=1 .LB \n(Li 0 1 4 \ give me a [1]
  .if \ng=2 .BL \ give me a bullet
  .if \ng=3 .LB \n(Li 0 2 2 a \ give me an a)
  .if \ng=4 .ML + \ give me a +
..

```

This macro can be used, in conjunction with `.LI` and `.LE`, instead of `.AL`, `.RL`, `.BL`, `.LB`, and `.ML`. For example, the following input:

```

.aL
.LI
first line.
.aL
.LI
second line.
.LE
.LI
third line.
.LE

```

will yield:

```

A. first line.
    [1] second line.
    [2] third line.

```

There is another approach to lists that is similar to the `.H` mechanism. The list-initialization, as well as the `.LI` and the `.LE` macros are all included in a single macro. That macro (called `.bL` below) requires an argument to tell it what level of item is required; it adjusts the list level by either beginning a new list or setting the list level back to a previous value, and then issues a `.LI` macro call to produce the item:


```
.de bL
.ie \\n(.$ .nr g \\$1 \ " if there is an argument, that is the level
.el .nr g \\n(:g \ " if no argument, use current level
.\ " can't increase level by more than 1
.if \\ng-\\n(:g>1 .)D " **ILLEGAL SKIPPING OF LEVEL
.if \\ng>\\n(:g \{.aL \\ng-1 \ " if g > :g, begin new list
.  nr g \\n(:g\} \ " and reset g to current level (.aL changes g)
.if \\n(:g>\\ng .LC \\ng \ " if :g > g, prune back to correct level
\ " if :g = g, stay within current list
.LI \ " in all cases, get out an item
..
```

For `.bL` to work, the previous definition of the `.aL` macro must be changed to obtain the value of `g` from its argument, rather than from `:g`. Invoking `.bL` without arguments causes it to stay at the current list level. The `-mm` `.LC` macro (List Clear) removes list descriptions until the level is less than or equal to that of its argument. For example, the `.H` macro includes the call `".LC 0"`. If text is to be resumed at the end of a list, insert the call `".LC 0"` to clear out the lists completely. The example below illustrates the relatively small amount of input needed by this approach. The input text:

```
The quick brown fox jumped over the lazy dog's back.
.bL 1
first line.
.bL 2
second line.
.bL 1
third line.
.bL
fourth line.
.LC 0
fifth line.
```

yields:

```
The quick brown fox jumped over the lazy dog's back.
      B. first line.
         [1] second line.
      C. third line.
      D. fourth line.
fifth line.
```

5.9 DISPLAYS

Displays are blocks of text that are to be kept together — not split across pages. `-mm` provides two styles of displays: a static (`.DS`) style and a floating (`.DF`) style. In the static style, the display

appears in the same relative position in the output text as it does in the input text; this may result in extra white space at the bottom of the page if the display is too big to fit there. In the floating style, the display “floats” through the input text to the top of the next page if there is not enough room for it on the current page; thus, the input text that follows a floating display may precede it in the output text. Since floating displays are kept in a queue, their relative order is not disturbed.

By default, a display is processed in no-fill mode, with single-spacing, and is not indented from the existing margins. You can specify indentation or centering, as well as fill-mode processing.

Displays and footnotes may never be nested, in any combination whatsoever. Although lists are permitted, no headings (.H or .HU) can occur within displays or footnotes.

5.9.1 Static Displays

```
.DS [format] [fill] [rindent]
one or more lines of text
.DE
```

A static display is started by the .DS macro and terminated by the .DE macro. With no arguments, .DS will accept the lines of text exactly as they are typed (no-fill mode) and will not indent them from the prevailing left margin indentation or from the right margin. The *rindent* (right indent) argument is the number of characters that the line length should be decreased, i.e., an indentation from the right margin. This number must be unscaled in **nroff** and is treated as *ens*. It may be scaled in **troff**. Otherwise, it defaults to *ems*.

The *format* argument to .DS is an integer or letter used to control the left margin indentation and centering with the following meanings:

<i>Code</i>	<i>Meaning</i>
" "	no indent
0 or L	no indent
1 or I	indent by standard amount
2 or C	center each line
3 or CB	center as a block

The *fill* argument is also an integer or letter and can have the following meanings:

<i>Code</i>	<i>Meaning</i>
" "	no-fill mode
0 or N	no-fill mode
1 or F	fill mode

Omitted arguments are taken to be zero.

The standard amount of indentation is taken from the register S_i , which is initially set to 5. Thus, by default, the text of an indented display aligns with the first line of indented paragraphs, whose indent is contained in the P_i register. Even though their initial values are the same, these two registers are independent of one another.

The display format value 3 (CB) centers the entire display as a block (as opposed to `.DS 2` and `.DF 2`, which center each line individually). That is, all the collected lines are left-justified, and, then, the display is centered, based on the width of the longest line. This format must be used in order for the `eqn[1]/neqn(1)` “mark” and “lineup” feature to work with centered equations.

By default, a blank line (one-half of a vertical space) is placed before and after static and floating displays. These blank lines before and after *static* displays can be inhibited by setting the register D_s to 0.

The following example shows the usage of all three arguments for displays. This block of text will be filled and indented 5 spaces from both the left and the right margins (i.e., centered).

```
.DS I F 5
```

```
“We the people of the United States, in order to form a more perfect union,
establish justice, ensure domestic tranquility, provide for the common defense,
and secure the blessings of liberty to ourselves and our posterity,
do ordain and establish this Constitution to the
United States of America.”
```

```
.DE
```

5.9.2 Floating Displays

```
.DF [format] [fill] [rindent]
one or more lines of text
.DE
```

A floating display begins with a `.DF` macro and ends with a `.DE` macro. The arguments have the same meanings as for `.DS`, except that, for floating displays, indent, no indent, and centering are always calculated with respect to the initial left margin, because the prevailing indent may change between the time when the formatter first reads the floating display and the time that the display is printed. The formatter always adds one blank line (one-half of a vertical space) before and after a floating display.

The user may exercise great control over the output positioning of floating displays through the use of two number registers, D_e and D_f . When a floating display is encountered by `nroff` or `troff`, it is processed and placed onto a queue of displays waiting to be output. Displays are always removed from the queue and printed in the

order that they were entered on the queue, which is the order that they appeared in the input file. If a new floating display is encountered and the queue of displays is empty, then the new display is a candidate for immediate output on the current page. Immediate output is governed by the size of the display and the setting of the *Df* register (see below). The *De* register (see below) controls whether or not text will appear on the current page after a floating display has been produced.

As long as the queue contains one or more displays, new displays will be automatically entered there, rather than being output. When a new page is started (or, when in two-column mode, the top of the second column is started), the next display from the queue becomes a candidate for output, if the *Df* register has specified “top-of-page” output. When a display is output it is also removed from the queue.

When the end of a section (when using section-page numbering) or the end of a document is reached, all displays are automatically removed from the queue and output. This will occur before a .SG, .CS, or .TC is processed.

A display is said to “fit on the current page” if there is enough room to contain the entire display on the page, or if the display is longer than one page in length and less than half of the current page has been used. Also, note that a wide (full page width) display will never fit in the second column of a two-column document.

The registers, their settings, and their effects are as follows:

Values for <i>De</i> Register	
Value	Action
0	DEFAULT: No special action occurs.
1	A page eject will <i>always</i> follow the output of each floating display, so only one floating display will appear on a page and no text will follow it.
	NOTE: For any other values the action performed is for the value 1.

Values for <i>Df</i> Register	
Value	Action
0	Floating displays will not be output until the end of the section (when section-page numbering) or end of document.
1	Output the new floating display on the current page if there is room, otherwise, hold it until the end of the section or document.
2	Output exactly one floating display from the queue at the top of a new page or column (when in two-column mode).
3	Output one floating display on current page if there is room. Output exactly one floating display at the top of a new page or column.
4	Output as many displays as will fit (at least one), starting at the top of a new page or column. Note that if register <i>De</i> is set to 1, each display will be followed by a page eject, causing a new top of page to be reached where at least one more display will be output. (This also applies to value 5, below.)
5	DEFAULT: Output a new floating display on the current page if there is room. Output at least one, but as many displays as will fit starting at the top of a new page or column. NOTE: For any value greater than 5, the action performed is for the value 5.

The `.WC` macro may also be used to control handling of displays in double-column mode and to control the break in the text before floating displays.

5.9.3 Tables

```
.TS [H]
global options;
column descriptors.
title lines
[.TH [N]]
data within the table.
.TE
```

The `.TS` (table start) and `.TE` (table end) macros make possible the use of the `tbl(1)` processor (see Chapter 5 of this section). They are used to delimit the text to be examined by `tbl[1]` as well as to set proper spacing around the table. The display function and the `tbl` delimiting function are independent of one another. To keep a block that contains any mixture of tables, equations, filled and unfilled text, and caption lines from being split at a page break, the `.TS-.TE` block should be enclosed within a display (`.DS-.DE`). Floating tables may be enclosed inside floating displays (`.DF-.DE`).

The macros `.TS` and `.TE` also permit the processing of tables that extend over several pages. If a table heading is needed for each

page of a multi-page table, specify the argument "H" to the `.TS` macro as above. Following the options and format information, the table heading is typed on as many lines as required and followed by the `.TH` macro. The `.TH` macro must occur when `".TS H"` is used. Note that this is not a feature of `tbl`, but of the macro definitions provided by `-mm`.

The table header macro `.TH` may take as an argument the letter `N`. This argument causes the table header to be printed only if it is the first table header on the page. This option is used when it is necessary to build long tables from smaller `.TS~ H/.TE` segments. For example,

```
.TS H
global options;
column descriptors.
Title lines
.TH
data
.TE
.TS H
global options;
column descriptors.
Title lines
.TH N
data
.TE
```

will cause the table heading to appear at the top of the first table segment, and no heading to appear at the top of the second segment when both appear on the same page. However, the heading will still appear at the top of each page of the table. This feature is used when a single table must be broken into segments because of table complexity (for example, too many blocks of filled text). If each segment had its own `.TS~ H/.TH` sequence, each segment would have its own header. However, if each table segment after the first uses `.TS~ H/.TH~ N` then the table header will only appear at the beginning of the table and the top of each new page or column that the table continues onto.

For `nroff`, the `-e` option (`-E` for `mm(1)`) may be used for terminals that are capable of finer printing resolution. This will cause better alignment of features such as the lines forming the corner of a box. Note that `-e` is not effective with `col[1]`.

5.9.4 Equations

```
.DS
.EQ [label]
equation(s)
.EN
.DE
```

The equation preprocessors `eqn[1]` and `neqn[1]` expect to use the `.EQ` (equation start) and `.EN` (equation end) macros as delimiters in the same way that `tbl` uses `.TS` and `.TE`; however, `.EQ` and `.EN` must occur inside a `.DS-DE` pair.

Note: There is an exception to this rule: if `.EQ` and `.EN` are used only to specify the delimiters for in-line equations or to specify `eqn/neqn` “defines,” `.DS` and `.DE` must not be used. If they are, extra blank lines will appear in the output.

The `.EQ` macro takes an argument that will be used as a label for the equation. By default, the label will appear at the right margin in the “vertical center” of the general equation. The `Eq` register may be set to 1 to change the labeling to the left margin.

The equation will be centered for centered displays; otherwise the equation will be adjusted to the opposite margin from the label.

5.9.5 Captions

```
.FG [title] [override] [flag]
.TB [title] [override] [flag]
.EC [title] [override] [flag]
.EX [title] [override] [flag]
```

The `.FG` (Figure Title), `.TB` (Table Title), `.EC` (Equation Caption) and `.EX` (Exhibit Caption) macros are normally used inside `.DS-DE` pairs to automatically number and title figures, tables, and equations. They use registers `Fg`, `Tb`, `Ec`, and `Ex`, respectively (see on `-rN5` to reset counters in sections). As an example, the call:

```
.FG "This is an illustration"
```

yields:

Figure 1. This is an illustration

`.TB` replaces “**Figure**” by “**TABLE**”; `.EC` replaces “**Figure**” by “**Equation**”, and `.EX` replaces “**Figure**” by “**Exhibit**”. Output is centered if it can fit on a single line; otherwise, all lines but the first are indented to line up with the first character of the title. The format of the numbers may be changed using the `.af` request of the formatter. The format of the caption may be changed from “**Figure 1. Title**” to “**Figure 1 - Title**” by setting the `Of` register to 1.

The *override* string is used to modify the normal numbering. If *flag* is omitted or 0, *override* is used as a prefix to the number; if *flag* is 1, *override* is used as a suffix; and if *flag* is 2, *override* replaces the number. If -rN5 is given, "section-figure" numbering is set automatically and the user-specified *override* string is ignored.

As a matter of style, table headings are usually placed ahead of the text of the tables, while figure, equation, and exhibit captions usually occur after the corresponding figures and equations.

5.9.6 List of Figures, Tables, Etc.

You may request that a List of Figures, List of Tables, List of Exhibits, and/or List of Equations be printed after the Table of Contents is printed by setting the number registers *Lf*, *Lt*, *Lx*, and *Le* (respectively) to 1. *Lf*, *Lt*, and *Lx* are 1 by default; *Le* is 0 by default.

The titles of these Lists may be changed by redefining the following strings which are shown here with their default values:

```
.ds Lf LIST OF FIGURES
.ds Lt LIST OF TABLES
.ds Lx LIST OF EXHIBITS
.ds Le LIST OF EQUATIONS
```

5.10 FOOTNOTES

There are two macros that delimit the text of footnotes, a string used to automatically number the footnotes, and a macro that specifies the style of the footnote text.

Note: Footnotes are processed in an environment that is different from that of the body of the text (see the *.ev* request).

5.10.1 Automatic Numbering of Footnotes

Footnotes may be automatically numbered by typing the three characters "*F" immediately after the text to be footnoted, without any intervening spaces. This will place the next sequential footnote number (in a smaller point size) a half-line above the text to be footnoted.

5.10.2 Delimiting Footnote Text

There are two macros that delimit the text of each footnote:

```
.FS [label]
one or more lines of footnote text
.FE
```

The *.FS* (footnote start) marks the beginning of the text of the footnote, and the *.FE* marks its end. The *label* on the *.FS*, if

present, will be used to mark the footnote text. Otherwise, the number retrieved from the string F will be used. Note that automatically-numbered and user-labeled footnotes may be intermixed. If a footnote is labeled (`.FS label`), the text to be footnoted *must* be followed by *label*, rather than by “*F”. The text between `.FS` and `.FE` is processed in fill mode. Another `.FS`, a `.DS`, or a `.DF` are not permitted between the `.FS` and `.FE` macros. Only labeled footnotes may be used with tables. Examples of this include:

1. Automatically-numbered footnote:

```
This is the line containing the word\*F
.FS
This is the text of the footnote.
.FE
to be footnoted.
```

2. Labelled footnote:

```
This is a labeled*
.FS *
The footnote is labeled with an asterisk.
.FE
footnote.
```

The text of the footnote (enclosed within the `.FS`-`.FE` pair) should immediately follow the word to be footnoted in the input text, so that “*F” or *label* occurs at the end of a line of input and the next line is the `.FS` macro call. It is also good practice to append an unpaddingable space to “*F” or *label* when they follow an end-of-sentence punctuation mark (i.e., period, question mark, exclamation point).

5.10.3 Format of Footnote Text

```
.FD [arg] [1]
```

Within the footnote text, you can control the formatting style by specifying text hyphenation, right margin justification, and text indentation, as well as left- or right-justification of the label when text indenting is used. The `.FD` macro is invoked to select the appropriate style. The first argument is a number from the left column of the following table. The formatting style for each number is given by the remaining four columns. For further explanation of the first two of these columns, see the definitions of the `.ad`, `.hy`, `.na`, and `.nh` requests (Chapter 2).

0	.nh	.ad	text indent	label left-justified
1	.hy	.ad	"	"
2	.nh	.na	"	"
3	.hy	.na	"	"
4	.nh	.ad	no text indent	"
5	.hy	.ad	"	"
6	.nh	.na	"	"
7	.hy	.na	"	"
8	.nh	.ad	text indent	label right-justified
9	.hy	.ad	"	"
10	.nh	.na	"	"
11	.hy	.na	"	"

If the first argument to `.FD` is out of range, the effect is as though `.FD 0` were specified. If the first argument is omitted or null, the effect is equivalent to `.FD 10` in **nroff** and to `.FD 0` in **troff**; these are also the respective initial defaults.

If a second argument is specified, then, whenever a first-level heading is encountered, automatically-numbered footnotes begin again with 1. This is most useful with the "section-page" page numbering scheme. As an example, the input line:

```
.FD " " 1
```

maintains the default formatting style and causes footnote numbers to be reset after each first-level heading.

For long footnotes that continue onto the following page, it is possible that, if hyphenation is permitted, the last line of the footnote on the current page will be hyphenated. Except for this case (which you can avoid by specifying an even argument to `.FD`), hyphenation across pages is inhibited by `-mm`.

Footnotes are separated from the body of the text by a short rule. Footnotes that continue to the next page are separated from the body of the text by a full-width rule. In **troff**, footnotes are set in type that is two points smaller than the point size used in the body of the text.

5.10.4 Spacing Between Footnote Entries

Normally, one blank line (a three-point vertical space) separates the footnotes when more than one occurs on a page. To change this spacing, set the register `Fs` to the desired value. For example,

```
.nr Fs 2
```

will cause two blank lines (a six-point vertical space) to occur between footnotes.

5.11 PAGE HEADERS AND FOOTERS

Text that occurs at the top of each page is known as the "\\page header." Text printed at the bottom of each page is called the

`\\page footer.`” The `-mm` package allows you to specify a header/footer that

- is printed on every page
- is printed on every odd-numbered page
- is printed on every even-numbered page

Thus, the page header may have up to two lines of text: the line that occurs at the top of every page and the line for the even- or odd-numbered page. The same is true for the page footer.

This section first describes the default appearance of page headers and page footers, and then the ways of changing them. We use the term `\\header`” (not qualified by `\\even`” or `\\odd`”) to mean the line of the page header that occurs on every page. We use the term `\\footer`” in the same way.

5.11.1 Default Headers and Footers

By default, each page has a centered page number as the header. There is no default footer and no even/odd default headers or footers.

5.11.2 Page Header

`.PH [arg]`

For this and for the `.EH`, `.OH`, `.PF`, `.EF`, `.OF` macros, the argument is of the form:

” ‘ left-part ‘ center-part ‘ right-part ‘ ”

If it is inconvenient to use the apostrophe (‘) as the delimiter (i.e., because it occurs within one of the parts), it may be replaced uniformly by any other character. On output, the parts are left-justified, centered, and right-justified, respectively.

The `.PH` macro specifies the header that is to appear at the top of every page. The initial value is the default centered page number enclosed by hyphens. The page number contained in the `P` register is an Arabic number. The format of the number may be changed by the `.af` request.

If debug mode is set using the flag `-rD1` on the command line, additional information, printed at the top left of each page, is included in the default header. This consists of the SCCS Release and Level of `-mm` (thus identifying the current version), followed by the current line number within the current input file.

5.11.3 Even-Page Header

`.EH [arg]`

The `.EH` macro supplies a line to be printed at the top of each even-numbered page, immediately following the header. The initial

value is a blank line.

5.11.4 Odd-Page Header

`.OH [arg]`

This macro is the same as `.EH`, except that it applies to odd-numbered pages.

5.11.5 Page Footer

`.PF [arg]`

The `.PF` macro specifies the line that is to appear at the bottom of each page. Its initial value is a blank line. If the `-rCn` flag is specified on the command line, the type of copy follows the footer on a separate line. In particular, if `-rC3` or `-rC4` (DRAFT) is specified, then, in addition, the footer is initialized to contain the date, instead of being a blank line.

5.11.6 Even-Page Footer

`.EF [arg]`

The `.EF` macro supplies a line to be printed at the bottom of each even-numbered page, immediately preceding the footer. The initial value is a blank line.

5.11.7 Odd-Page Footer

`.OF [arg]`

This macro is the same as `.EF`, except that it applies to odd-numbered pages.

5.11.8 Footer on the First Page

By default, the footer is a blank line. If, in the input text, you specify `.PF` and/or `.OF` before the end of the first page of the document, then these lines will appear at the bottom of the first page.

If the `-rN1` flag is specified on the command line, the header (whatever its contents) replaces the footer on the first page only.

5.11.9 Section-Page Numbering

Pages can be numbered sequentially within sections. To obtain this numbering style, specify `-rN3` or `-rN5` on the command line. In this case, the default *footer* is a centered "section-page" number, e.g., 7-2, and the default page header is blank.

5.11.10 Strings and Registers in Headers/Footers

String and register names may be placed in the arguments to the header and footer macros. If the value of the string or register is to be computed when the respective header or footer is printed, the invocation must be escaped by four (4) backslashes. This is because the string or register invocation will be processed three times:

- as the argument to the header or footer macro;
- in a formatting request within the header or footer macro;
- in a `.tl` request during header or footer processing.

For example, the page number register *P* must be escaped with four backslashes in order to specify a header in which the page number is to be printed at the right margin, e.g.,

```
.PH " ' ' ' Page \\\nP ' "
```

creates a right-justified header containing the word "Page" followed by the page number. Similarly, to specify a footer with the "section-page" style, specify:

```
.PF " ' ' ' - \\\n(H1-\\n - ' "
```

As another example, suppose you have arranged for the string */C* to contain the current section heading which is to be printed at the bottom of each page. The `.PF` macro call would then be:

```
.PF " ' ' \\\n*(/C ' ' "
```

If only one or two backslashes had been used, the footer would print a constant value for */C*, namely, its value when the `.PF` appeared in the input text.

5.11.11 Header and Footer Example

The following sequence specifies blank lines for the header and footer lines, page numbers on the outside edge of each page (i.e., top left margin of even pages and top right margin of odd pages), and "Revision 3" on the top inside margin of each page:

```
.PH " "
.PF " "
.EH " ' \\\nP ' ' Revision 3 ' '
.OH " ' Revision 3 ' ' \\\nP ' "
```

5.11.12 Generalized Top-of-Page Processing

Note: This section is intended only for those who are experienced in writing formatter macros.

During header processing, `-mm` invokes two user-definable macros. One, the `.TP` macro, is invoked in the environment (see `.ev` request) of the header; the other, `.PX`, is a user-exit macro that is invoked (without arguments) when the normal environment has been restored, and with "no-space" mode already in effect.

The effective initial definition of `.TP` (after the first page of a document) is:

```
.de TP
.sp 3
.tl \\*(}t
.if e `tl \\*(}e
.if o `tl \\*(}o
.sp 2
..
```

The string }*t* contains the header, the string }*e* contains the even-page header, and the string }*o* contains the odd-page header, as defined by the .PH, .EH, and .OH macros, respectively. To obtain more specialized page titles, the user may redefine the .TP macro to cause any desired header processing. Note that formatting done within the .TP macro is processed in an environment different from that of the body.

For example, to obtain a page header that includes three centered lines of data, say, a document's number, issue date, and revision date, one could define .TP as follows:

```
.de TP
.sp
.ce 3
777-888-999
Iss. 2, AUG 1977
Rev. 7, SEP 1977
.sp
..
```

The .PX macro may be used to provide text that is to appear at the top of each page after the normal header and that may have tab stops to align it with columns of text in the body of the document.

5.11.13 Generalized Bottom-of-Page Processing

```
.BS
zero or more lines of text
.BE
```

Lines of text that are specified between the .BS (bottom-block start) and .BE (bottom-block end) macros will be printed at the bottom of each page, after the footnotes (if any), but before the page footer. This block of text is removed by specifying an empty block, i.e.,

```
.BS
.BE
```

5.11.14 Top and Bottom Margins

```
.VM [top] [bottom]
```

`.VM` (Vertical Margin) allows the user to specify extra space at the top and bottom of the page. This space precedes the page header and follows the page footer. `.VM` takes two unscaled arguments that are treated as *v*'s. For example,

```
.VM 10 15
```

adds 10 blank lines to the default top of page margin, and 15 blank lines to the default bottom of page margin. Both arguments must be positive (default spacing at the top of the page may be decreased by re-defining `.TP`).

5.11.15 Private Documents

```
.nr Pv value
```

The word "PRIVATE" may be printed, centered, and underlined on the second line of a document (preceding the page header). This is done by setting the *Pv* register:

Value	Meaning
<code>.nr Pv 0</code>	do not print PRIVATE (default)
<code>.nr Pv 1</code>	PRIVATE on first page only
<code>.nr Pv 2</code>	PRIVATE on all pages

If *Pv* is 2, the user definable `.TP` may not be used because `.TP` is used by `-mm` to print PRIVATE on all pages except the first page of a memorandum on which `.TP` is not invoked.

5.12 TABLE OF CONTENTS AND COVER SHEET

The table of contents and the cover sheet for a document are produced by invoking the `.TC` and `.CS` macros, respectively.

These macros should normally appear only once at the end of the document, after the Signature Block macros. They may occur in either order.

The table of contents is produced at the end of the document because the entire document must be processed before the table of contents can be generated. Similarly, the cover sheet is often not needed, and is, therefore, produced at the end.

5.12.1 Table of Contents

```
.TC [slevel] [spacing] [tlevel] [tab] [head1] [head2] [head3] [head4] [head5]
```

The `.TC` macro generates a table of contents containing the headings that were saved for the table of contents as determined by the value of the *Cl* register. The arguments to `.TC` control the spacing before each entry, the placement of the associated page number, and additional text on the first page of the table of contents before the word "CONTENTS."

Spacing before each entry is controlled by the first two arguments; headings whose level is less than or equal to *slevel* will have *spacing* blank lines (halves of a vertical space) before them. Both *slevel* and *spacing* default to 1. This means that first-level headings are preceded by one blank line (one-half of a vertical space). Note that *slevel* does *not* control what levels of heading have been saved; the saving of headings is the function of the *Cl* register.

The third and fourth arguments control the placement of the page number for each heading. The page numbers can be justified at the right margin with either blanks or dots (“leaders”) separating the heading text from the page number, or the page numbers can follow the heading text. For headings whose level is less than or equal to *tlevel* (default 2), the page numbers are justified at the right margin. In this case, the value of *tab* determines the character used to separate the heading text from the page number. If *tab* is 0 (the default value), dots (i.e., leaders) are used; if *tab* is greater than 0, spaces are used. For headings whose level is greater than *tlevel*, the page numbers are separated from the heading text by two spaces (i.e., they are “ragged right”).

All additional arguments (e.g., *head1*, *head2*, etc.), if any, are horizontally centered on the page, and precede the actual table of contents itself.

If the *.TC* macro is invoked with at most four arguments, then the user-exit macro *.TX* is invoked (without arguments) before the word “CONTENTS” is printed, or the user-exit macro *.TY* is invoked and the word “CONTENTS” is *not* printed. By defining *.TX* or *.TY* and invoking *.TC* with at most four arguments, the user can specify what needs to be done at the top of the (first) page of the table of contents. For example, the following input:

```
.de TX
.ce 2
Special Application
Message Transmission
.sp 2
.in +10n
Approved: \l' 3i'
.in
.sp
..
.TC
```

yields:

Special Application
Message Transmission

Approved: _____

CONTENTS

⋮

If this macro were defined as `.TY` rather than `.TX`, the word "CONTENTS" would not appear. Defining `.TY` as an empty macro will suppress "CONTENTS" with no replacement:

```
.de TY
..
```

By default, the first level headings will appear in the table of contents at the left margin. Subsequent levels will be aligned with the text of headings at the preceding level. These indentations may be changed by defining the *Ci* string which takes a maximum of seven arguments corresponding to the heading levels. It must be given at least as many arguments as are set by the *Cl* register. The arguments must be scaled. For example, with *Cl*=5,

```
.ds Ci .25i .5i .75i 1i 1i
```

or

```
.ds Ci 0 2n 4n 6n 8n
```

Two other registers are available to modify the format of the table of contents, *Oc* and *Cp*. By default, table of contents pages will have lowercase Roman numeral page numbering. If the *Oc* register is set to 1, the `.TC` macro will not print any page number but will instead reset the *P* register to 1. In this case, you must supply an appropriate page footer to replace the page number. Ordinarily, the same `.PF` used in the body of the document (e.g., OSDD style) will be adequate.

The List of Figures, Tables, etc. pages will be produced separately, unless *Cp* is set to 1, which causes these lists to appear on the same page as the table of contents.

5.13 REFERENCES

There are two macros that delimit the text of references, a string used to automatically number the references, and an optional macro to produce reference pages within the document.

5.13.1 Automatic Numbering of References

Automatically numbered references may be obtained by typing `*(Rf` immediately after the text to be referenced. This places the

next sequential reference number (in a smaller point size) enclosed in brackets a half-line above the text to be referenced.

5.13.2 Delimiting Reference Text

The `.RS` and `.RF` macros are used to delimit text for each reference.

```
A line of text to be referenced.\*(Rf
.RS [string-name]
reference text
.RF
```

5.13.3 Subsequent References

`.RS` takes one argument, a *string-name*. For example,

```
.RS AA
reference text
.RF
```

The string `AA` is assigned the current reference number. It may be used later in the document, as the string call, `*(AA`, to reference text which must be labeled with a prior reference number. The reference is output enclosed in brackets a half-line above the text to be referenced. No `.RS/.RF` is needed for subsequent references.

5.13.4 Reference Page

An automatically generated reference page is produced at the end of the document before the Table of Contents and the Cover Sheet are output. The reference page is entitled "References". This page contains the reference text (`RS/RF`). The user may change the Reference Page title by defining the *Rp* string. For example,

```
.ds Rp "New Title"
```

The optional `.RP` (Reference Page) macro may be used to produce reference pages anywhere within a document (i.e., within heading sections).

```
.RP [arg1] [arg2]
```

These arguments allow the user to control resetting of reference numbering, and page skipping.

arg1	Meaning
0	reset reference counter (default)
1	do not reset reference counter
arg2	Meaning
0	cause a following <code>.SK</code> (default)
1	do not cause a following <code>.SK</code>

`.RP` need not be used unless you wish to produce reference pages

elsewhere in the document.

5.14 MISCELLANEOUS FEATURES

5.14.1 Bold, Italic, and Roman Type

```
.B [bold-arg] [previous-font-arg] ...
.I [italic-arg] [previous-font-arg] ...
.R
```

When called without arguments, `.B` changes the font to bold and `.I` changes to underlining (Italic). This condition continues until the occurrence of a `.R`, when the regular roman font is restored. Thus,

```
.I
  here is some text.
.R
```

yields:

here is some text.

If `.B` or `.I` is called with one argument, that argument is printed in the appropriate font (underlined in **nroff** for `.I`). Then, the *previous* font is restored (underlining is turned off in **nroff**). If two or more arguments (maximum 6) are given to a `.B` or `.I`, the second argument is then concatenated to the first with no intervening space (1/12 space if the first font is Italic), but is printed in the previous font; the remaining pairs of arguments are similarly alternated. For example,

```
.I Italic " text " right -justified
```

produces:

Italic text right-justified

These macros alternate with the prevailing font at the time they are invoked. To alternate specific pairs of fonts, the following macros are available:

```
.IB
.BI
.IR
.RI
.RB
.BR
```

Each takes a maximum of 6 arguments and alternates the arguments between the specified fonts.

Note that font changes in headings are handled separately.

If you are using a terminal that cannot underline, you might wish to insert:

```
.rm ul
.rmcu
```

at the beginning of the document to eliminate *all* underlining.

5.14.2 Justification of Right Margin

```
.SA [arg]
```

The `.SA` macro is used to set right-margin justification for the main body of text. Two justification flags are used: *current* and *default*. `.SA 0` sets both flags to no justification, i.e., it acts like the `.na` request. `.SA 1` is the inverse: it sets both flags to cause justification, just like the `.ad` request. However, calling `.SA` *without* an argument causes the *current* flag to be copied from the *default* flag, thus performing either a `.na` or `.ad`, depending on the *default*. Initially, both flags are set for no justification in **nroff** and for justification in **troff**.

In general, the request `.na` can be used to ensure that justification is turned. To restore justification, use `.SA`, rather than the `.ad` request. That way, justification (or lack thereof) for the remainder of the text is specified by inserting `.SA 0` or `.SA 1` *once* at the beginning of the document.

5.14.3 SCCS Release Identification

The string *RE* contains the SCCS Release and Level of the current version of **-mm**. For example, typing:

```
This document was processed using version \*(RE of the macros.
produces:
```

```
This document was processed using version 15.110 of the macros.
```

This information is useful in analyzing suspected bugs in **-mm**. The easiest way to have this number appear in your output is to specify `-rD1` on the command line, which causes the string *RE* to be output as part of the page header {9.2}.

5.14.4 Two-Column Output

Mm can print two columns on a page:

```
.2C
text and formatting requests (except another .2C)
.1C
```

The `.2C` macro begins two-column processing, which continues until a `.1C` macro is encountered. In two-column processing, each physical page is thought of as containing two columnar “pages” of equal (but smaller) “page” width. Page headers and footers are *not* affected by two-column processing. The `.2C` macro does *not* “balance” two-column output.

It is possible to have full-page width footnotes and displays when in two column mode, although the default action is for footnotes and displays to be narrow in two column mode and wide in one column mode. Footnote and display width is controlled by a macro, **.WC** (Width Control), which takes the following arguments:

N	Normal default mode (-WF, -FF, -WD, FB)
WF	Wide Footnotes always (even in two column mode)
-WF	DEFAULT: turn off WF (footnotes follow column mode, wide in 1C mode, narrow in 2C mode, unless FF is set)
FF	First Footnote; all footnotes have the same width as the <i>first</i> footnote encountered for that page
-FF	DEFAULT: turn off FF (footnote style follows the settings of WF or -WF)
WD	Wide Displays always (even in two column mode)
-WD	DEFAULT: Displays follow whichever column mode is in effect when the display is encountered
FB	DEFAULT: Floating displays cause a break when output on the current page
-FB	Floating displays on current page do not cause a break

For example, **.WC WD FF** will cause all displays to be wide, and all footnotes on a page to be the same width, while **.WC N** will reinstate the default actions. If conflicting settings are given to **.WC** the last one is used. That is, **.WC WF -WF** has the effect of **.WC -WF**.

5.14.5 Column Headings

In two-column output, it is sometimes necessary to have headers over each column, as well as headers over the entire page. This is accomplished by redefining the **.TP** macro to provide header lines both for the entire page and for each of the columns. For example,

```
.de TP
.sp 2
.tl 'Page \\nP' OVERALL ' '
.tl ' ' TITLE ' '
.sp
.nf
.ta 16C 31R 34 50C 65R
left ⊕ center ⊕ right ⊕ left ⊕ center ⊕ right
```

(where ⊕ stands for the tab character)

```
⊕ first column ⊕ ⊕ ⊕ second column
.fi
.sp 2
..
```

The above example will produce two lines of page header text plus two lines of headers over each column. The tab stops are for a 65-en overall line length.

5.14.6 Vertical Spacing

```
.SP [lines]
```

There are several ways of obtaining vertical spacing, all with different effects.

The `.sp` request spaces the number of lines specified, *unless* “no space” (`.ns`) mode is on, in which case the request is ignored. This mode is typically set at the end of a page header in order to eliminate spacing by a `.sp` or `.bp` request that just happens to occur at the top of a page. This mode can be turned *off* using the `.rs` (“restore spacing”) request.

The `.SP` macro is used to avoid the accumulation of vertical space by successive macro calls. Several `.SP` calls in a row produce *not* the sum of their arguments, but their maximum; i.e., the following produces only 3 blank lines:

```
.SP 2
.SP 3
.SP
```

Many `-mm` macros utilize `.SP` for spacing. For example, “.LE 1” immediately followed by “.P” produces only a single blank line (one-half of a vertical space) between the end of the list and the following paragraph. An omitted argument defaults to one blank line (*one* vertical space). Negative arguments are not permitted. The argument must be unscaled, but fractional amounts are permitted. Like `.sp`, `.SP` is also inhibited by the `.ns` request.

5.14.7 Skipping Pages

`.SK [pages]`

The `.SK` macro skips pages, but retains the usual header and footer processing. If *pages* is omitted, null, or 0, `.SK` skips to the top of the next page *unless* it is currently at the top of a page, in which case it does nothing. `.SK n` skips *n* pages. That is, `.SK` always positions the text that follows it at the top of a page, while `.SK 1` always leaves one page that is blank except for the header and footer.

5.14.8 Forcing an Odd Page

`.OP`

This macro is used to ensure that subsequent output text begins at the top of an odd-numbered page. If output is currently at the top of an odd page, no motion takes place. If output is currently on an even page, text resumes printing at the top of the next page; if output is now on an odd page (but not at the top of the page) one blank page is produced, and printing resumes on the page after that.

5.14.9 Setting Point Size and Vertical Spacing

In **troff**, the default point size (obtained from the register *S* is 10, with a vertical spacing of 12 points (i.e., 6 lines per inch). The prevailing point size and vertical spacing may be changed by invoking the `.S` macro:

`.S [point size] [vertical spacing]`

The mnemonics, D for default value, C for current value, and P for previous value, may be used for both point size and vertical spacing arguments.

Arguments may be signed or unsigned. If an argument is negative, the current value is decremented by the specified amount. If the argument is positive, the current value is incremented by the specified amount. If an argument is unsigned, it is used as the new value. `.S` without arguments defaults to previous (P). If the first argument is specified but the second argument (vertical spacing) is not, then the default (D) value is used. The default value for vertical spacing is always 2p greater than the current point size value selected.

Note: Footnotes are printed two points *smaller* than the body copy point size. An additional vertical spacing of three points is placed between footnotes.

A null (") argument for either the first or second argument defaults to the current (C) value. For example, (where *n* is a

numeric value),

```
.S          = .S P P
.S "" n    = .S C n
.S n ""    = .S n C
.S n       = .S n D
.S ""      = .S C D
.S "" ""   = .S C C
.S n n     = .S n n
```

If a point size argument is greater than 99, the default point size (D) 10 is restored. If a vertical spacing argument is greater than 99, the default vertical spacing (D) +2p is used. For example,

```
.S 12 111 = .S 12 14
.S 110     = .S 10 12
```

5.14.10 Producing Accents

The following strings may be used to produce diacritical marks (accents):

	Input	Output
Grave accent	a*'	à
Acute accent	a*'	á
Circumflex	a*	â
Tilde	n*~	ñ
Cedilla	c*,	ç
Lower-case umlaut	a*:	ä
Upper-case umlaut	A*;	Ä

5.14.11 Inserting Text Interactively

```
.RD [prompt] [diversion] [string]
```

.RD (ReaD insertion) tells the formatter to stop reading the input file and, instead, read text from the standard input until two consecutive newlines are found. When the newlines are encountered, the formatter resumes processing the input file at the point where it had been stopped by .RD.

.RD follows the formatting conventions in effect. Thus, the examples below assume that the .RD is invoked in no fill mode (.nf).

The first argument is a *prompt* which will be printed at the terminal. If no prompt is given, .RD prompts with a BEL on terminal output.

The second argument, a *diversion* name, allows you to save all text typed after the prompt. The third argument, a *string* name, allows the user to save for later reference the first line following the

prompt. For example,

```
.RD Name aa bb
```

produces

```
Name: (user types) J. Jones
16 Elm Rd.,
Piscataway
```

The diversion *aa* will contain:

```
J. Jones
16 Elm Rd.,
Piscataway
```

The string *bb* (`*(bb)`) contains "J. Jones".

A newline followed by a Control D (EOF) also allows the user to resume normal output.

5.14.12 Bell Labs Macros

The `-mm` package includes a number of macros that are used only at Bell Labs facilities. They produce papers in various Bell Labs formats, and are not of general interest to other UNIX users. Included under this heading are

- `.TL` (Title)
- `.AU` (Author)
- `.TM` (A BTL Internal Document Number)
- `.OK` (Other Keywords)
- `.MT` (Type of Memorandum)
- `.AS` (Abstract Start)
- `.AE` (Abstract End)

These macros are well-known to Bell Labs personnel and will not be further explained here.

5.14.13 Date and Format Changes

```
.ND new-date
```

The `.ND` macro alters the value of the string *DT*, which is initially set to the current date. If you do not reset this string, the current date will always be interpolated wherever `\(DT` is placed in an input file.

5.14.14 “Copy to” and Other Notations

```
.NS [arg]
zero or more lines of the notation
.NE
```

Various “copy to” notations are obtained through the .NS macro, which provides for the proper spacing and for breaking the notations across pages, if necessary.

The codes for *arg* and the corresponding notations are:

<i>Code</i>	<i>Notations</i>
.NS ””	Copy to
.NS 0	Copy to
.NS	Copy to
.NS 1	Copy (with att.) to
.NS 2	Copy (without att.) to
.NS 3	Att.
.NS 4	Atts.
.NS 5	Enc.
.NS 6	Encs.
.NS 7	Under Separate Cover
.NS 8	Letter to
.NS 9	Memorandum to
.NS ” <i>string</i> ”	Copy (<i>string</i>) to

If *arg* consists of more than one character, it is placed within parentheses between the words “Copy” and “to.” For example,

```
.NS ”with att. 1 only”
```

will generate “Copy (with att. 1 only) to” as the notation. More than one notation may be specified before the .NE occurs, because a .NS macro terminates the preceding notation, if any. For example,

```
.NS 4
Attachment 1-List of register names
Attachment 2-List of string and macro names
.NS 1
R. M. Mottola
.NS 2
J. R. Reilly
R. M. Horton
.NE
```

would be formatted as:

Atts.

Attachment 1-List of register names

Attachment 2-List of string and macro names

Copy (with att.) to

R. M. Mottola

Copy (without att.) to

J. R. Reilly

R. M. Horton

5.14.15 Approval Signature Line

.AV approver's-name

The .AV macro may be used after the last notation block to automatically generate a line with spaces for the approval signature and date. For example,

.AV "Jane Doe"

produces:

APPROVED:

Jane Doe

Date

5.14.16 Forcing a One-Page Letter

To force an increase in the page length (so that a letter or memo can be made to fit on a single page), use the `-rLn` option, e.g. `-rL90`. This has the effect of making the formatter believe that the page is 90 lines long, and that therefore it has more room than usual to place the signature or the notations. This will *only* work for a *single-page* letter or memo.

5.15 ERRORS AND DEBUGGING

5.15.1 Error Terminations

When a macro discovers an error, the following actions occur:

- A break occurs.
- To avoid confusion regarding the location of the error, the formatter output buffer (which may contain some text) is printed.
- A short message is printed giving the name of the macro that found the error, the type of error, and the approximate line number (in the current input file) of the last processed input line.
- Processing terminates, unless the register *D* has a positive value. In the latter case, processing continues even though the output

is guaranteed to be garbled from that point on.

Note: The error message is written on the transcript pad of the window in which **troff** is running. If you are using an output filter, the message may be garbled by being intermixed with text held in that filter's output buffer. If either **tbl** or **n/eqn** or both are being used, and if the **-olist** option of the formatter causes the last page of the document not to be printed, a harmless "broken pipe" message results.

5.15.2 Disappearance of Output

This usually occurs because of an unclosed diversion (e.g., missing **.FE** or **.DE**). Fortunately, the macros that use diversions are careful about it, and they check to make sure that illegal nestings do not occur. If any message is issued about a missing **.DE** or **.FE**, the appropriate action is to search backwards from the termination point looking for the corresponding **.DS**, **.DF**, or **.FS**.

The following command:

```
grep -n " \.[EDFT][EFNQS]" files ...
```

prints all the **.DS**, **.DF**, **.DE**, **.FS**, **.FE**, **.TS**, **.TE**, **.EQ**, and **.EN** macros found in *files ...*, each preceded by its file name and the line number in that file. This listing can be used to check for illegal nesting and/or omission of these macros.

5.15.3 MM Error Messages

Each **-mm** error message consists of a standard part followed by a variable part. The standard part is of the form:

ERROR:input line *n* :

The variable part consists of a descriptive message, usually beginning with a macro name. The variable parts are listed below in alphabetical order by macro name, each with a more complete explanation:

AL:bad arg:value The argument to the **.AL** macro is not one of 1, A, a, I, or i. The incorrect argument is shown as *value*.

CS:cover sheet too long The text of the cover sheet is too long to fit on one page. The abstract should be reduced or the indent of the abstract should be decreased.

DS:too many displays More than 26 floating displays are active at once, i.e., have been accumulated but not yet output.

DS:missing FE	A display starts inside a footnote. The likely cause is the omission (or misspelling) of a .FE to end a previous footnote.
DS:missing DE	.DS or .DF occurs within a display, i.e., a .DE has been omitted or mistyped.
DE:no DS or DF active	.DE has been encountered but there has not been a previous .DS or .DF to match it.
FE:no FS	.FE has been encountered with no previous .FS to match it.
FS:missing FE	A previous .FS was not matched by a closing .FE, i.e., an attempt is being made to begin a footnote inside another one.
FS:missing DE	A footnote starts inside a display, i.e., a .DS or .DF occurs without a matching .DE.
H:bad arg:value	The first argument to .H must be a single digit from 1 to 7, but <i>value</i> has been supplied instead.
H:missing FE	A heading macro (.H or .HU) occurs inside a footnote.
H:missing DE	A heading macro (.H or .HU) occurs inside a display.
H:missing arg	.H needs at least 1 argument.
HU:missing arg	.HU needs 1 argument.
LB:missing arg(s)	.LB requires at least 4 arguments.
LB:too many nested lists	Another list was started when there were already 6 active lists.
LE:mismatched	.LE has occurred without a previous .LB or other list-initialization macro. Although this is not a fatal error, the message is issued because there almost certainly exists some problem in the preceding text.
LI:no lists active	.LI occurs without a preceding list-initialization macro. The latter has probably been omitted, or has been separated from the .LI by an intervening .H or .HU.
ML:missing arg	.ML requires at least 1 argument.
ND:missing arg	.ND requires 1 argument.
SA:bad arg:value	The argument to .SA (if any) must be either 0 or 1. The incorrect argument is shown as <i>value</i> .

SG:missing DE	.SG occurs inside a display.
SG:missing FE	.SG occurs inside a footnote.
SG:no authors	.SG occurs without any previous .AU macro(s).
VL:missing arg	.VL requires at least 1 argument.

5.15.4 Formatter Error Messages

Most messages issued by the formatter are self-explanatory. Those error messages over which you may have some control are listed below. Any other error messages should be reported to your system administrator.

“Cannot do ev” is caused by *(a)* setting a page width that is negative or extremely short, *(b)* setting a page length that is negative or extremely short, *(c)* reprocessing a macro package (e.g., requesting, via .so a macro package that was also requested from the command line), and *(d)* requesting the `-sl` option to *troff* on a document that is longer than ten pages.

“Cannot execute *filename*” is given by the `!` request if it cannot find the *filename*.

“Cannot open *filename*” is issued if one of the files in the list of files to be processed cannot be opened.

“Exception word list full” indicates that too many words have been specified in the hyphenation exception list (via .hw requests).

“Line overflow” means that the output line being generated was too long for the formatter’s line buffer. The excess was discarded. See the “Word overflow” message below.

“Non-existent font type” means that a request has been made to mount an unknown font.

“Non-existent macro file” means that the requested macro package does not exist.

“Non-existent terminal type” means that the terminal options refer to an unknown terminal type.

“Out of temp file space” means that additional temporary space for macro definitions, diversions, etc. cannot be allocated. This message often occurs because of unclosed diversions (missing .FE or .DE), unclosed macro definitions (e.g., missing “.”), or a huge table of contents.

“Too many page numbers” is issued when the list of pages specified to the formatter `-o` option is too long.

“Too many string/macro names” is issued when the pool of string and macro names is full. Unneeded strings and macros can be

deleted using the `.rm` request.

“Too many number registers” means that the pool of number register names is full. Unneeded registers can be deleted by using the `.rr` request.

“Word overflow” means that a word being generated exceeded the formatter’s word buffer. The excess characters were discarded. A likely cause for this and for the “Line overflow” message above are very long lines or words generated through the misuse of `\c` or of the `.cu` request, or very long equations produced by `eqn(1)/neqn(1)`.

5.16 EXTENDING AND MODIFYING THE MACROS

5.16.1 Naming Conventions

In this section, the following conventions are used to describe legal names:

n:	digit
a:	lower-case letter
A:	upper-case letter
x:	any letter or digit (any alphanumeric character)
s:	special character (any non-alphanumeric character)

All other characters are literals (i.e., stand for themselves).

Note that *request*, *macro*, and *string* names are kept by the formatters in a single internal table, so that there must be no duplication among such names. *Number register* names are kept in a separate table.

5.16.2 Names Used by Formatters

requests:	aa (most common)
	an (only one, currently: <code>.c2</code>)
registers:	aa (normal)
	<code>.x</code> (normal)
	<code>.s</code> (only one, currently: <code>.\$</code>)
	<code>%</code> (page number)

5.16.3 Names Used by `-mm`

macros:	AA (most common, accessible to user)
	A (less common, accessible to user)
	<code>]x</code> (internal, constant)
	<code>>x</code> (internal, dynamic)
strings:	AA (most common, accessible to user)
	A (less common, accessible to user)
	<code>]x</code> (internal, usually allocated to specific functions throughout)
	<code>}x</code> (internal, more dynamic usage)

registers: Aa (most common, accessible to users)
 An (common, accessible to user)
 A (accessible, set on command line)
 :x (mostly internal, rarely accessible, usually dedi-
 cated)
 ;x (internal, dynamic, temporaries)

5.16.4 Names Used by eqn, neqn, and tbl

The equation preprocessors, **eqn**(1) and **neqn**(1), use registers and string names of the form *nn*. The table preprocessor, **tbl**(1), uses names of the form:

a- a+ a| nn #a ## #- # a T. TW

5.16.5 User-Definable Names

To avoid problems, we suggest using names that consist either of a single lower-case letter, or of a lower-case letter followed by anything other than a lower-case letter. The following is a sample naming convention:

macros: aA
 Aa

strings: a
 a) (or a], or a}, etc.)

registers a
 aA

5.17 SAMPLE EXTENSIONS

5.17.1 Appendix Headings

The following gives a way of generating and numbering appendices:

```
.nr Hu 1
.nr a 0
.de aH
.nr a +1
.nr P 0
.PH " ' ' ' Appendix \\na - \\\n"
.SK
.HU "\\$1"
..
```

After the above initialization and definition, each call of the form ".aH "title"" begins a new page (with the page header changed to "Appendix *a - n*") and generates an unnumbered heading of *title*, which, if desired, can be saved for the table of contents. Those who wish Appendix titles to be centered must, in addition, set the register *Hc* to 1.

5.17.2 Hanging Indent with Tabs

The following example illustrates the use of the hanging-indent feature of variable-item lists. First, a user-defined macro is built to accept four arguments that make up the *mark*. Each argument is to be separated from the previous one by a tab character; tab settings are defined later. Since the first argument may begin with a period or apostrophe, the “\.” is used so that the formatter will not interpret such a line as a formatter request or macro.

Note: The two-character sequence “\.” is understood by the formatters to be a “zero-width” space; i.e., it causes no output characters to appear.

The “\t” is translated by the formatter into a tab character. The “\c” is used to concatenate the line of *text* that follows the macro to the line of text built by the macro. The macro definition and an example of its use are as follows:

```
.de aX
.LI
.\||$1\t||$2\t||$3\t||$4\t\c
..
:
.ta 9n 18n 27n 36n
.VL 36
.aX .nh off \- no
No hyphenation.
Automatic hyphenation is turned off.
Words containing hyphens
(e.g., mother-in-law) may still be split across lines.
.aX .hy on \- no
Hyphenate.
Automatic hyphenation is turned on.
.aX .hc\  c none none no          (  stands for a space)
Hyphenation indicator character is set to “c” or removed.
During text processing the indicator is suppressed
and will not appear in the output.
Prepending the indicator to a word has the effect
of preventing hyphenation of that word.
.LE
```

The resulting output is:

.nh	off	-	no	No hyphenation. Automatic hyphenation is turned off. Words containing hyphens (e.g., mother-in-law) may still be split across lines.
.hy	on	-	no	Hyphenate. Automatic hyphenation is turned on.
.hc c	none	none	no	Hyphenation indicator character is set to "c" or removed. During text processing the indicator is suppressed and will not appear in the output. Prepending the indicator to a word has the effect of preventing hyphenation of that word.

5.18 SUMMARY OF MACROS, STRINGS, AND NUMBER REGISTERS

5.18.1 Macros

The following is an alphabetical list of macro names used by **-mm**. The first line of each item gives the name of the macro and a brief description. The second line gives a prototype call of the macro.

Macros marked with an asterisk are *not*, in general, invoked directly by the user. Rather, they are "user exits" called from inside header, footer, or other macros.

1C	One-column processing .1C
2C	Two-column processing .2C
AE	Abstract end .AE
AF	Alternate format of "Subject/Date/From" block .AF [company-name]
AL	Automatically-incremented list start .AL [type] [text-indent] [1]
AS	Abstract start .AS [arg] [indent]
AT	Author's title .AT [title] ...
AU	Author information .AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]

AV	Approval signature .AV [name]
B	Bold .B [bold-arg] [previous-font-arg] [bold] [prev] [bold] [prev]
BE	Bottom End .BE
BI	Bold/Italic .BI [bold-arg] [italic-arg] [bold] [italic] [bold] [italic]
BL	Bullet list start .BL [text-indent] [1]
BR	Bold/Roman .BR [bold-arg] [Roman-arg] [bold] [Roman] [bold] [Roman]
BS	Bottom Start .BS
CS	Cover sheet .CS [pages] [other] [total] [figs] [tbls] [refs]
DE	Display end .DE
DF	Display floating start .DF [format] [fill] [right-indent]
DL	Dash list start .DL [text-indent] [1]
DS	Display static start .DS [format] [fill] [right-indent]
EC	Equation caption .EC [title] [override] [flag]
EF	Even-page footer .EF [arg]
EH	Even-page header .EH [arg]
EN	End equation display .EN
EQ	Equation display start .EQ [label]
EX	Exhibit caption .EX [title] [override] [flag]
FC	Formal closing .FC [closing]

FD	Footnote default format .FD [arg] [1]
FE	Footnote end .FE
FG	Figure title .FG [title] [override] [flag]
FS	Footnote start .FS [label]
H	Heading—numbered .H level [heading-text] [heading-suffix]
HC	Hyphenation character .HC [hyphenation-indicator]
HM	Heading mark style (Arabic or Roman numerals, or letters) .HM [arg1] ... [arg7]
HU	Heading—unnumbered .HU heading-text
HX *	Heading user exit X (before printing heading) .HX dlevel rlevel heading-text
HY *	Heading user exit Y (before printing heading) .HY dlevel rlevel heading-text
HZ *	Heading user exit Z (after printing heading) .HZ dlevel rlevel heading-text
I	Italic (underline in nroff) .I [italic-arg] [previous-font-arg] [italic] [prev] [italic] [prev]
IB	Italic/Bold .IB [italic-arg] [bold-arg] [italic] [bold] [italic] [bold]
IR	Italic/Roman .IR [italic-arg] [Roman-arg] [italic] [Roman] [italic] [Roman]
LB	List begin .LB text-indent mark-indent pad type [mark] [LI-space] [LB-space]
LC	List-status clear .LC [list-level]
LE	List end .LE [1]
LI	List item .LI [mark] [1]
ML	Marked list start .ML mark [text-indent] [1]

MT	Memorandum type .MT [type] [addressee] <i>or</i> .MT [4] [1]
ND	New date .ND new-date
NE	Notation end .NE
NS	Notation start .NS [arg]
nP	Double-line indented paragraphs .nP
OF	Odd-page footer .OF [arg]
OH	Odd-page header .OH [arg]
OK	Other keywords for TM cover sheet .OK [keyword] ...
OP	Odd page .OP
P	Paragraph .P [type]
PF	Page footer .PF [arg]
PH	Page header .PH [arg]
PM	Proprietary Marking .PM [code]
PX *	Page-header user exit .PX
R	Return to regular (roman) font (end underlining in nroff) .R
RB	Roman/Bold .RB [Roman-arg] [bold-arg] [Roman] [bold] [Roman] [bold]
RD	Read insertion from terminal .RD [prompt] [diversion] [string]
RF	Reference end .RF
RI	Roman/Italic .RI [Roman-arg] [italic-arg] [Roman] [italic] [Roman] [italic]

RL	Reference list start .RL [text-indent] [1]
RP	Produce Reference Page .RP [arg] [arg]
RS	Reference start .RS [string-name]
S	Set <i>troff</i> point size and vertical spacing .S [size] [spacing]
SA	Set adjustment (right-margin justification) default .SA [arg]
SG	Signature line .SG [arg] [1]
SK	Skip pages .SK [pages]
SP	Space—vertically .SP [lines]
TB	Table title .TB [title] [override] [flag]
TC	Table of contents .TC [slevel] [spacing] [tlevel] [tab] [head1] [head2] [head3] [head4] [head5]
TE	Table end .TE
TH	Table header .TH [N]
TL	Title of memorandum .TL [charging-case] [filing-case]
TM	Technical Memorandum number(s) .TM [number] ...
TP *	Top-of-page macro .TP
TS	Table start .TS [H]
TX *	Table-of-contents user exit .TX
TY *	Table-of-contents user exit (suppresses “CONTENTS”) .TY
VL	Variable-item list start .VL text-indent [mark-indent] [1]

VM Vertical margins
.VM [top] [bottom]

WC Width Control
.WC [format]

5.18.2 Strings

The following is an alphabetical list of string names used by **-mm**; for each, there is a brief description, section reference, and initial (default) value(s).

BU Bullet
nroff: ⊕
troff: •

Ci Contents indent up to seven args for heading levels (must be scaled)

F Footnote numberer
nroff: \u\\n+(;p\d
troff: \v'-.4m'\s-3\\n+(;p\s0\v'.4m'

DT Date (current date, unless overridden)
Month day, year (e.g., May 28, 1985)

EM Em dash string, produces an em dash for both **nroff** and **troff**

HF Heading font list, up to seven codes for heading levels 1 through 7
3 3 2 2 2 2 2 (levels 1 and 2 bold, 3-7 underlined in **nroff**, and italic in **troff**)

HP Heading point size list, up to seven codes for heading levels 1 through 7

Le Title for LIST OF EQUATIONS

Lf Title for LIST OF FIGURES

Lt Title for LIST OF TABLES

Lx Title for LIST OF EXHIBITS

RE SCCS Release and Level of **-mm**
Release.Level (e.g., 15.110)

Rf Reference numberer

Rp Title for References

Tm Trademark string places the letters "TM" one half-line above the text that it follows

Note that if the released-paper style is used, then, in addition to the above strings, certain BTL location codes are defined as strings; these location strings are needed only until the **.MT** macro is called.

Also accent strings are available.

5.18.3 Number Registers

This section provides an alphabetical list of register names; for each, a brief description, section reference, initial (default) value, and the legal range of values (where [m:n] means values from m to n inclusive) are provided.

Any register having a single-character name can be set from the command line. An asterisk attached to a register name indicates that that register can be set *only* from the command line or *before* the **-mm** macro definitions are read by the formatter.

- A * Handles preprinted forms and the Bell Logo
0, [0:2]
- Au Inhibits printing of author's location, department, room,
and extension in the "from" portion of a memorandum
1, [0:1]
- C * Copy type (Original, DRAFT, etc.)
0 (Original), [0:4]
- Cl Contents level (i.e., level of headings saved for table of
contents)
2, [0:7]
- Cp Placement of List of Figures, etc.
1 (on separate pages), [0:1]
- D * Debug flag
0, [0:1]
- De Display eject register for floating displays
0, [0:1]
- Df Display format register for floating displays
5, [0:5]
- Ds Static display pre- and post-space
1, [0:1]
- Ec Equation counter, used by .EC macro
0, [0:?], incremented by 1 for each .EC call.
- Ej Page-ejection flag for headings
0 (no eject), [0:7]
- Eq Equation label placement
0 (right-adjusted), [0:1]
- Ex Exhibit counter, used by .EX macro
0, [0:?], incremented by 1 for each .EX call.
- Fg Figure counter, used by .FG macro
0, [0:?], incremented by 1 for each .FG call.

Fs	Footnote space (i.e., spacing between footnotes) 1, [0:?]
H1-H7	Heading counters for levels 1-7 0, [0:?], incremented by .H of corresponding level or .HU if at level given by register <i>Hu</i> . H2-H7 are reset to 0 by any heading at a lower-numbered level.
Hb	Heading break level (after .H and .HU) 2, [0:7]
Hc	Heading centering level for .H and .HU 0 (no centered headings), [0:7]
Hi	Heading temporary indent (after .H and .HU) 1 (indent as paragraph), [0:2]
Hs	Heading space level (after .H and .HU) 2 (space only after .H 1 and .H 2), [0:7]
Ht	Heading type (for .H: single or concatenated numbers) 0 (concatenated numbers: 1.1.1, etc.), [0:1]
Hu	Heading level for unnumbered heading (.HU) 2 (.HU at the same level as .H 2), [0:7]
Hy	Hyphenation control for body of document 0 (automatic hyphenation off), [0:1]
L *	Length of page 66, [20:?] (11i, [2i:?] in troff) ⁻¹
Le	List of Equations 0 (list not produced) [0:1]
Lf	List of Figures 1 (list produced) [0:1]
Li	List indent 6, [0:?]
Ls	List spacing between items by level 5 (spacing between all levels) [0:5]
Lt	List of Tables 1 (list produced) [0:1]
Lx	List of Exhibits 1 (list produced) [0:1]
N *	Numbering style 0, [0:5]
Np	Numbering style for paragraphs 0 (unnumbered) [0:1]

- O * Offset of page
 .75i, [0:?] (0.5i, [0i:?] in **troff**)⁻¹
- Oc Table of Contents page numbering style
 0 (lowercase Roman), [0:1]
- Of Figure caption style
 0 (period separator), [0:1]
- P Page number, managed by **-mm**
 0, [0:?]
- Pi Paragraph indent
 5, [0:?]
- Ps Paragraph spacing
 1 (one blank space between paragraphs), [0:?]
- Pt Paragraph type
 0 (paragraphs always left-justified), [0:2]
- Pv “PRIVATE” header
 0 (not printed), [0:2]
- S * **Troff** default point size
 10, [6:36]
- Si Standard indent for displays
 5, [0:?]
- T * Type of **nroff** output device
 0, [0:2]
- Tb Table counter
 0, [0:?], incremented by 1 for each .TB call.
- U * Underlining style (**nroff**) for .H and .HU
 0 (continuous underline when possible), [0:1]
- W * Width of page (line and title length)
 6i, [10:1365] (6i, [2i:7.54i] in **troff**)⁻¹

Chapter 6: Eqn — a Pre-Processor for Text With Equations

6.1 INTRODUCTION

Eqn is a preprocessor designed to allow people who know neither mathematics nor typesetting to typeset mathematical equations easily. Enough of the language to set in-line expressions like $\lim_{x \rightarrow \pi/2} (\tan x)^{\sin 2x} = 1$ or display equations, like the ones, below can be learned in an hour or so.

$$\begin{aligned} G(z) &= e^{\ln G(z)} = \exp\left(\sum_{k \geq 1} \frac{S_k z^k}{k}\right) = \prod_{k \geq 1} e^{S_k z^k / k} \\ &= \left(1 + S_1 z + \frac{S_1^2 z^2}{2!} + \dots\right) \left(1 + \frac{S_2 z^2}{2} + \frac{S_2^2 z^4}{2^2 \cdot 2!} + \dots\right) \dots \\ &= \sum_{m \geq 0} \left[\sum_{\substack{k_1, k_2, \dots, k_m \geq 0 \\ k_1 + 2k_2 + \dots + mk_m = m}} \frac{S_1^{k_1}}{1^{k_1} k_1!} \frac{S_2^{k_2}}{2^{k_2} k_2!} \dots \frac{S_m^{k_m}}{m^{k_m} k_m!} \right] z^m \end{aligned}$$

Eqn interfaces directly with **troff**, so mathematical expressions can be embedded in the running text of a manuscript, allowing the entire document to be produced in one process.

Eqn input files may be used with **nroff** as well. The input is identical, but you have to use a variant of **eqn**, called **neqn** instead. Of course, some things won't be as attractive because terminals don't provide the variety of characters, sizes, and fonts that a typesetter does, but the output is usually adequate for proofreading. **Eqn** knows relatively little about mathematics. In particular, mathematical symbols like +, -, ×, parentheses, and so on do not have any special meanings.

Eqn is normally invoked on the **troff** command line, using a statement like

```
eqn file(s) troff option(s)
```

6.2 DISPLAYED EQUATIONS

To tell **eqn** where a mathematical expression begins, use the **.EQ** macro. Enter the text of the equation, then end it with a **.EN** macro. Thus, if you type the lines

```
.EQ
x=y+z
.EN
```

your output will look like

$$x=y+z$$

The `.EQ` and `.EN` are copied through untouched; they are not processed by `eqn`. This means that you have to take care of centering, numbering, and so on. The most common way is to use a macro package like `-ms` or `-mm`, which will number equations and display them in various ways.

With the `-ms` package, equations are centered by default. To left-justify an equation, use `.EQ L` instead of `.EQ`. To indent it, use `.EQ I`. Any of these can be followed by an arbitrary "equation number" which will be placed at the right margin. For example, the input

```
.EQ I (3.1a)
x = f(y/2) + y/2
.EN
```

produces the output

$$x=f(y/2)+y/2 \qquad (3.1a)$$

There is also a shorthand notation so that in-line expressions like π_i^2 can be entered without `.EQ` and `.EN`.

Note: When using `eqn` with the `-mm` macro package, always force a break, using the `.br` request, before the `.EQ` macro.

6.3 SPACES AND NEWLINES

6.3.1 Input Spaces

Spaces and newlines within an expression are thrown away by `eqn`. Normal text is left alone. Thus, between `.EQ` and `.EN`,

$$x=y+z$$

and

$$x = y + z$$

and

$$x = y \\ + z$$

all produce the same output,

$$x=y+z$$

Use spaces and newlines freely to make your input equations readable and easy to edit. In particular, very long lines are a bad idea, since, with some editors, they are hard to fix if you make a mistake.

6.3.2 Output Spaces

To force extra spaces into the output, use a tilde “ ~ ” for each space you want:

$$x \sim = \sim y \sim + \sim z$$

gives

$$x = y + z$$

You can also use a circumflex “^”, which gives a space half the width of a tilde. It is mainly useful for fine-tuning. Tabs may also be used to position pieces of an expression, but the tab stops must be set by **troff** commands.

6.4 SYMBOLS, SPECIAL NAMES, GREEK

Eqn knows some mathematical symbols, some mathematical names, and the Greek alphabet. For example,

$$x=2 \text{ pi int sin (omega t)dt}$$

produces

$$x=2\pi \int \sin(\omega t) dt$$

Here the spaces in the input are necessary. They tell **eqn** that *int*, *pi*, *sin*, and *omega* are separate entities that should get special treatment. The *sin*, digit 2, and parentheses are set in roman type instead of italic; *pi* and *omega* are made Greek; and *int* becomes the integral sign.

When in doubt, leave spaces around separate parts of the input. A common error is to type $f(\text{pi})$ without leaving spaces on both sides of the *pi*. As a result, **eqn** does not recognize *pi* as a special word, and it appears as $f(\text{pi})$ instead of $f(\pi)$.

6.5 DELIMITING SPECIAL SEQUENCES

The only way **eqn** can deduce that some sequence of letters might be special is if that sequence is separated from the letters on either side of it. This can be done by surrounding a special word by ordinary spaces (or tabs or newlines), as we did in the previous section.

You can also make special words stand out by surrounding them with tildes or circumflexes:

$$x \sim = \sim 2 \sim \text{ pi } \sim \text{ int } \sim \text{ sin } \sim (\sim \text{ omega } \sim \text{ t } \sim) \sim \text{ dt}$$

is much the same as the last example, except that the tildes not only separate the magic words like *sin*, *omega*, and so on, but also add extra spaces, one space per tilde:

$$x = 2 \pi \int \sin (\omega t) dt$$

Special words can also be separated by braces { } and double quotes "...", which have special meanings.

6.6 SUBSCRIPTS AND SUPERSCRIPTS

Subscripts and superscripts are obtained with the words *sub* and *sup*.

$$x \text{ sup } 2 + y \text{ sub } k$$

gives

$$x^2 + y_k$$

eqn takes care of all the size changes and vertical motions needed to make the output look right. The words *sub* and *sup* must be surrounded by spaces; *x sub2* will give you *xsub2* instead of *x₂*. Furthermore, don't forget to leave a space (or a tilde, etc.) to mark the end of a subscript or superscript. A common error is to say something like

$$y = (x \text{ sup } 2) + 1$$

which causes

$$y = (x^2) + 1$$

instead of the intended

$$y = (x^2) + 1$$

Subscripted subscripts and superscripted superscripts also work:

$$x \text{ sub } i \text{ sub } 1$$

is

$$x_{i_1}$$

A subscript and superscript on the same thing are printed one above the other if the subscript comes *first*:

$$x \text{ sub } i \text{ sup } 2$$

is

$$x_i^2$$

Other than this special case, *sub* and *sup* group to the right, so *x sup y sub z* means *x^y_z*, not *x^{y_z}*.

6.7 BRACES FOR GROUPING

Normally, the end of a subscript or superscript is marked simply by a blank (or tab, or tilde, etc.). If the subscript or superscript is something that has to be typed with blanks in it, use braces to mark the beginning and end of the subscript or superscript:

$$e \sup \{i \text{ omega } t\}$$

is

$$e^{i\omega t}$$

As a rule, braces can *always* be used to force **eqn** to treat something as a unit, or just to make your intent perfectly clear. Thus,

$$x \text{ sub } \{i \text{ sub } 1\} \text{ sup } 2$$

is

$$x_{i_1}^2$$

with braces, but

$$x \text{ sub } i \text{ sub } 1 \text{ sup } 2$$

is

$$x_{i_1}^2$$

which is rather different.

Braces can be nested within other sets of braces if necessary:

$$e \sup \{i \text{ pi } \sup \{\rho + 1\}\}$$

is

$$e^{i\pi^{\rho+1}}$$

The general rule is that anywhere you could use some single character like x , you can use an arbitrarily complicated expression if you enclose it in braces. **Eqn** will look after all the details of positioning it and making it the right size.

In all cases, make sure you have the right number of braces. Leaving one out or adding an extra will cause **eqn** to generate error messages.

Occasionally, you will have to print braces in your output. To do this, enclose them in double quotes, like "{". Quoting is discussed in more detail in a later section of this chapter.

6.8 FRACTIONS

To make a fraction, use the word *over*:

$$a+b \text{ over } 2c = 1$$

gives

$$\frac{a+b}{2c} = 1$$

The line is made the right length and positioned automatically. Braces can be used to make clear what goes over what:

{alpha + beta} over {sin (x)}

is

$$\frac{\alpha+\beta}{\sin(x)}$$

When there is both an *over* and a *sup* in the same expression, **eqn** does the *sup* before the *over*, so

-b sup 2 over pi

is $-\frac{b^2}{\pi}$ instead of $-b^{\frac{2}{\pi}}$. The rules that decide which operation is done first in cases like this are summarized later in this chapter. When in doubt, use braces to make clear what parts of the expression should be treated together.

6.9 SQUARE ROOTS

To draw a square root, use *sqrt*:

sqrt a+b + 1 over sqrt {ax sup 2 +bx+c}

is

$$\sqrt{a+b} + \frac{1}{\sqrt{ax^2+bx+c}}$$

Square roots of tall quantities may not look very good when set. This is because a root-sign big enough to cover the quantity is too dark and heavy:

sqrt {a sup 2 over b sub 2}

is

$$\sqrt{\frac{a^2}{b_2}}$$

Big square roots are generally better written as something to the power $\frac{1}{2}$.

6.10 SUMMATION, INTEGRAL, ETC.

Summations, integrals, and similar constructions are easy:

sum from i=0 to {i= inf} x sup i

produces

$$\sum_{i=0}^{i=\infty} x^i$$

Notice that we used braces to indicate where the upper part $i=\infty$ begins and ends. No braces were necessary for the lower part $i=0$, because it contained no blanks. The braces will never hurt, and if the *from* and *to* parts contain any blanks, you must use braces

around them.

The *from* and *to* parts are both optional, but if both are used, they have to occur in that order.

Other useful characters can replace the *sum* in our example:

int prod union inter

become, respectively,

\int \prod \cup \cap

Since the thing before the *from* can be anything, even something in braces, *from-to* can often be used in unexpected ways:

lim from {n -> inf} x sub n = 0

is

$\lim_{n \rightarrow \infty} x_n = 0$

6.11 SIZE AND FONT CHANGES

By default, equations are set in 10-point type with standard mathematical conventions to determine which characters are in roman and which in italic. Although **eqn** makes an attempt to use pleasing sizes and fonts, it is not perfect. To change sizes and fonts, use *size n* and *roman*, *italic*, *bold* and *fat*. Like *sub* and *sup*, size and font changes affect only the character or complete expression that follows them, and revert to the normal situation at the end of it. Thus

bold x y

is

xy

and

size 14 bold x = y +
size 14 {alpha + beta}

gives

x=y+α+β

As always, you can use braces if you want to affect something more complicated than a single letter. For example, you can change the size of an entire equation by

size 12 { ... }

Legal sizes which may follow *size* are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, 36. You can also change the size *by* a given amount; for example, you can say *size +2* to make the size two points bigger, or *size -3* to make it three points smaller.

If you are using fonts other than roman, italic and bold, you can say *font X* where *X* is a one character **troff** name or number for the font. Since **eqn** is tuned for Roman, Italic, and bold, other fonts may not give quite as attractive an appearance.

The *fat* operation takes the current font and widens it by overstriking: *fat grad* is ∇ and *fat {x sub i}* is x_i .

If an entire document is to be in a non-standard size or font, it is a serious nuisance to have to write out a size and font change for each equation. Accordingly, you can set a "global" size or font which affects all equations thereafter. At the beginning of any equation, you might say, for instance,

```
.EQ
gsize 16
gfont R
...
.EN
```

to set the size to 16 and the font to roman. In place of R, you can use any of the **troff** font names. The size after *gsize* can be a relative change with + or -.

Generally, *gsize* and *gfont* will appear at the beginning of a document but they can also appear throughout a document; the global font and size can be changed as often as needed. For example, in a footnote[‡] you will typically want the size of equations to match the size of the footnote text, which is two points smaller than the main text. Don't forget to reset the global size at the end of the footnote.

6.12 DIACRITICAL MARKS

To get accent and other diacritical marks on top of letters, there are several words:

x dot	\dot{x}
x dotdot	\ddot{x}
x hat	\hat{x}
x tilde	\tilde{x}
x vec	\vec{x}
x dyad	\overleftrightarrow{x}
x bar	\bar{x}
x under	\underline{x}

The diacritical mark is placed at the right height. The *bar* and

[‡]Like this one, in which we have a few random expressions like x_i and π^2 . The sizes for these were set by the command *gsize -2*.

under are made the right length for the entire construct, as in $\overline{x+y+z}$; other marks are centered.

6.13 QUOTED TEXT

Any input entirely within quotes ("...") is not subject to any of the font changes and spacing adjustments normally done by the equation setter. This provides a way to do your own spacing and adjusting if needed:

italic "sin(x)" + sin (x)

is

sin(x)+sin(x)

Quotes are also used to get braces and other **eqn** keywords printed:

"{ size alpha }"

is

{ *size alpha* }

and

roman "{ size alpha }"

is

{ size alpha }

The construction "" is often used as a place-holder when **eqn** needs something grammatically, but you don't actually want anything in your output. For example, to make ²He, you can't just type *sup 2 roman He* because a *sup* has to have something to put the superscript above. Thus, you must say

"" sup 2 roman He

To get a literal quote, use "\". **Troff** characters like $\langle bs$ can appear unquoted, but more complicated things like horizontal and vertical motions with $\langle h$ and $\langle v$ should always be quoted.

6.14 LINING UP EQUATIONS

It's sometimes necessary to line up a series of equations at some horizontal position, often at an equals sign. This is done with two operations called *mark* and *lineup*.

The word *mark* may appear once at any place in an equation. It remembers the horizontal position where it appeared. Successive equations can contain one occurrence of the word *lineup*. The place where *lineup* appears is made to line up with the place marked by the previous *mark*, if at all possible. Thus, for example, you can say

```
.EQ I
x+y mark = z
.EN
.EQ I
x lineup = 1
.EN
```

to produce

```
x+y=z
x=1
```

When you use **eqn** and ‘-ms’, use either **.EQ I** or **.EQ L**, since **mark** and **lineup** don’t work with centered equations. Also, bear in mind that **mark** doesn’t look ahead;

```
x mark =1
...
x+y lineup =z
```

isn’t going to work, because there isn’t room for the **x+y** part after the **mark** remembers where the **x** is.

6.15 LARGE DELIMITERS

To get big brackets [], braces { }, parentheses (), and bars around things, use the **left** and **right** commands:

```
left { a over b + 1 right }
~ = ~ left ( c over d right )
+ left [ e right ]
```

is

$$\left\{ \frac{a}{b} + 1 \right\} = \left(\frac{c}{d} \right) + [e]$$

The resulting brackets are made big enough to cover whatever they enclose. Other characters can be used besides these, but they are not likely to look very good. One exception is the **floor** and **ceiling** characters:

```
left floor x over y right floor
<= left ceiling a over b right ceiling
```

produces

$$\left\lfloor \frac{x}{y} \right\rfloor \leq \left\lceil \frac{a}{b} \right\rceil$$

Several warnings about brackets are in order. First, braces are typically bigger than brackets and parentheses, because they are made up of three, five, seven, etc., pieces, while brackets can be made up of two, three, etc.

The *right* part may be omitted: a “left something” need not have a corresponding “right something”. If the *right* part is omitted, put braces around the thing you want the left bracket to encompass. Otherwise, the resulting brackets may be too large.

If you want to omit the *left* part, things are more complicated, because technically you can't have a *right* without a corresponding *left*. Instead you have to say

left ”” right)

for example. The *left* ”” means a “left nothing”. This satisfies the rules without hurting your output.

6.16 PILES

There is a general facility for making vertical piles of things; it comes in several formats. For example,

$$\begin{array}{l} A \sim = \sim \text{ left } [\\ \quad \text{pile } \{ a \text{ above } b \text{ above } c \} \\ \quad \sim \sim \text{ pile } \{ x \text{ above } y \text{ above } z \} \\ \text{right }] \end{array}$$

will make

$$A = \begin{bmatrix} a & x \\ b & y \\ c & z \end{bmatrix}$$

The elements of the pile (there can be as many as you want) are centered one above another, at the right height for most purposes. The keyword *above* is used to separate the pieces; braces are used around the entire list. The elements of a pile can be as complicated as needed, even containing more piles.

Three other forms of pile exist: *lpile* makes a pile with the elements left-justified; *rpile* makes a right-justified pile; and *cpile* makes a centered pile, just like *pile*. The vertical spacing between the pieces is somewhat larger for *l-*, *r-* and *cpiles* than it is for ordinary piles.

$$\begin{array}{l} \text{roman sign } (x) \sim = \sim \\ \text{left } \{ \\ \quad \text{lpile } \{ 1 \text{ above } 0 \text{ above } -1 \} \\ \quad \sim \sim \text{ lpile} \\ \quad \{ \text{if } \sim x > 0 \text{ above if } \sim x = 0 \text{ above if } \sim x < 0 \} \end{array}$$

makes

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Notice the left brace without a matching right one.

6.17 MATRICES

It is also possible to make matrices. For example, to make a neat array like

$$\begin{array}{l} x_i \quad x^2 \\ y_i \quad y^2 \end{array}$$

you have to type

```
matrix {
  ccol { x sub i above y sub i }
  ccol { x sup 2 above y sup 2 }
}
```

This produces a matrix with two centered columns. The elements of the columns are then listed just as for a pile, each element separated by the word *above*. You can also use *lcol* or *rcol* to left or right adjust columns. Each column can be separately adjusted, and there can be as many columns as you like.

The reason for using a matrix instead of two adjacent piles, by the way, is that if the elements of the piles don't all have the same height, they won't line up properly. A matrix forces them to line up, because it looks at the entire structure before deciding what spacing to use.

Note: When using matrices, be sure that each column has the same number of elements in it. If you don't, **eqn** will do unpredictable things.

6.18 SHORTHAND FOR IN-LINE EQUATIONS

In a mathematical document, it is necessary to follow mathematical conventions not just in display equations, but also in the body of the text, for example by making variable names like *x* italic.

Although this could be done by surrounding the appropriate parts with **.EQ** and **.EN**, the continual repetition of **.EQ** and **.EN** is a nuisance. Furthermore, with '-ms', **.EQ** and **.EN** imply a displayed equation.

Eqn provides a shorthand for short in-line expressions. You can define two characters to mark the left and right ends of an in-line equation, and then type expressions right in the middle of text lines. To set both the left and right characters to dollar signs, for example, add to the beginning of your document the three lines

```
.EQ
delim $$
.EN
```

Having done this, you can then say things like

Let α_i be the primary variable, and let β be zero. Then we can show that x_1 is ≥ 0 .

This works as you might expect — spaces, newlines, and so on are significant in the text, but not in the equation part itself. Multiple equations can occur in a single input line.

Enough room is left before and after a line that contains in-line expressions that something like $\sum_{i=1}^n x_i$ does not interfere with the lines surrounding it.

To turn off the delimiters,

```
.EQ
delim off
.EN
```

Note: Don't use braces, tildes, circumflexes, or double quotes as delimiters — chaos will result.

6.19 DEFINITIONS

Eqn provides a facility so you can give a frequently-used string of characters a name, and thereafter just type the name instead of the whole string. For example, if the sequence

$$x_{i1} + y_{i1}$$

appears repeatedly throughout a paper, you can save re-typing it each time by defining it like this:

```
define xy 'xi1 + yi1'
```

This makes **xy** a shorthand for whatever characters occur between the single quotes in the definition. You can use any character instead of quote to mark the ends of the definition, so long as it doesn't appear inside the definition.

Now you can use **xy** like this:

```
.EQ
f(x) = xy ...
.EN
```

and so on. Each occurrence of **xy** will expand into what it was defined as. Be careful to leave spaces or their equivalent around the name when you actually use it, so **eqn** will be able to identify it as special.

There are several things to watch out for. First, although definitions can use previous definitions, as in

```
.EQ
define xi 'x sub i '
define xil 'xi sub 1 '
.EN
```

you cannot define something in terms of itself. A common error is to say

```
define X 'roman X '
```

This guarantees disaster, since *X* is now defined in terms of itself. If, instead, you say

```
define X 'roman "X" '
```

the quotes protect the second X, and everything works fine.

Eqn keywords can be redefined. You can make / mean *over* by saying

```
define / 'over '
```

or redefine *over* as / with

```
define over '/ '
```

If you need to make different things print on a terminal than on the typesetter, it is sometimes worth defining a symbol differently in **neqn** and **eqn**. This can be done with *ndefine* and *tdefine*. A definition made with *ndefine* only takes effect if you are running **neqn**. If you use *tdefine*, the definition only applies for **eqn**. Names defined with plain *define* apply to both **eqn** and **neqn**.

6.20 LOCAL MOTIONS

Although **eqn** tries to get most things at the right place on the paper, it isn't perfect, and occasionally, you will need to tune the output to make it just right. Small extra horizontal spaces can be obtained with tilde and circumflex. You can also say *back n* and *fwd n* to move small amounts horizontally. *n* is how far to move in 1/100's of an em. Thus, *back 50* moves back half an em. (See Chapter 1 of this section for more information on ems). Similarly, you can move things up or down with *up n* and *down n*. As with *sub* or *sup*, the local motions affect the next thing in the input, and this can be something arbitrarily complicated if it is enclosed in braces.

6.21 A LARGE EXAMPLE

Here is the complete source for the three display equations used in the introduction to this chapter.


```
.EQ
G(z) ~ mark = ~ e sup { ln ~ G(z) }
~ = ~ exp left (
sum from k >= 1 { S sub k z sup k } over k right )
~ = ~ prod from k >= 1 e sup { S sub k z sup k / k }
.EN
.EQ
lineup = left ( 1 + S sub 1 z +
{ S sub 1 sup 2 z sup 2 } over 2! + ... right )
left ( 1 + { S sub 2 z sup 2 } over 2
+ { S sub 2 sup 2 z sup 4 } over { 2 sup 2 cdot 2! }
+ ... right ) ...
.EN
.EQ
lineup = sum from m >= 0 left (
sum from
pile { k sub 1 , k sub 2 , ..., k sub m >= 0
above
k sub 1 + 2k sub 2 + ... + mk sub m = m }
{ S sub 1 sup { k sub 1 } } over { 1 sup k sub 1 k sub 1 ! } ~
{ S sub 2 sup { k sub 2 } } over { 2 sup k sub 2 k sub 2 ! } ~
...
{ S sub m sup { k sub m } } over { m sup k sub m k sub m ! }
right ) z sup m
.EN
```

6.22 KEYWORDS, PRECEDENCES, ETC.

If you don't use braces, **eqn** will do operations in the order shown in this list.

dyad vec under bar tilde hat dot dotdot
fwd back down up
fat roman italic bold size
sub sup sqrt over
from to

These operations group to the left:

over sqrt left right

All others group to the right.

Digits, parentheses, brackets, punctuation marks, and these mathematical words are converted to Roman font when encountered:

sin cos tan sinh cosh tanh arc
max min lim log ln exp
Re Im and if for det

These character sequences are recognized and translated as shown.

>=	>
<=	<
==	≡

\neq	\neq
\pm	\pm
\rightarrow	\rightarrow
\leftarrow	\leftarrow
\ll	\ll
\gg	\gg
inf	∞
partial	∂
half	
prime	'
approx	\approx
nothing	
cdot	\cdot
times	\times
del	∇
grad	∇
...	\dots
,...,	,:...',
sum	Σ
int	\int
prod	Π
union	\cup
inter	\cap

To obtain Greek letters, simply spell them out in whatever case you want:

DELTA	Δ	iota	ι
GAMMA	Γ	kappa	κ
LAMBDA		Λ lambda	λ
OMEGA	Ω	mu	μ
PHI	Φ	nu	ν
PI	Π	omega	ω
PSI	Ψ	omicron	\omicron
SIGMA	Σ	phi	ϕ
THETA	Θ	pi	π
UPSILON		Υ psi	ψ
XI	Ξ	rho	ρ
alpha	α	sigma	σ
beta	β	tau	τ
chi	χ	theta	θ
delta	δ	upsilon	υ
epsilon	ϵ	xi	ξ
eta	η	zeta	ζ
gamma	γ		

These are all the words known to **eqn** (except for characters with names).

above	lpile
back	mark
bar	matrix
bold	ndefine
ccol	over
col	pile
cpile	rcol
define	right
delim	roman
dot	rpile
dotdot	size
down	sqrt
dyad	sub
fat	sup
font	tdefine
from	tilde
fwd	to
gfont	under
gsize	up
hat	vec
italic	~ , ^
lcol	{ }
left	"..."
lineup	

6.23 TROUBLESHOOTING

If you make a mistake in an equation, like leaving out a brace (very common) or having one too many (also very common) or having a *sup* with nothing before it (common), **eqn** will tell you with the message

syntax error between lines x and y, file z

where *x* and *y* are the approximate lines between which the trouble occurred, and *z* is the name of the file in question. The line numbers may not specify the exact point of the error; look nearby as well. There are also self-explanatory messages that arise if you leave out a quote or try to run **eqn** on a non-existent file.

If you want to check a document before actually printing it, you can always throw away the output by redirection, as in

```
eqn files >/dev/null
```

If you use something like dollar signs as delimiters, it is easy to leave one out. This causes very strange troubles. The program **checkeq[1]** checks for misplaced or missing dollar signs and similar

troubles.

In-line equations can only be so big because of an internal buffer in **troff**. If you get a message “word overflow”, you have exceeded this limit. If you print the equation as a displayed equation, this message will usually go away. The message “line overflow” indicates you have exceeded an even bigger buffer. The only cure for this is to break the equation into two separate ones.

On a related topic, **eqn** does not break equations by itself; you must split long equations up across multiple lines by yourself, marking each by a separate **.EQEN** sequence. **eqn** does warn about equations that are too long to fit on one line.

Chapter 7: Tbl — a Preprocessor for Formatting Tables

7.1 INTRODUCTION

Tbl is a preprocessor that makes all sorts of tables easy to specify and enter. Tables are made up of columns which may be independently centered, right-adjusted, left-adjusted, or aligned by decimal points. Headings may be placed over single columns or groups of columns. A table entry may contain equations, or may consist of several rows of text. Horizontal or vertical lines may be drawn as desired in the table, and any table or element may be enclosed in a box. For example,

1970 Federal Budget Transfers (in billions of dollars)			
State	Taxes collected	Money spent	Net
New York	22.91	21.35	-1.56
New Jersey	8.33	6.96	-1.37
Connecticut	4.12	3.10	-1.02
Maine	0.74	0.67	-0.07
California	22.29	22.42	+0.13
New Mexico	0.70	1.49	+0.79
Georgia	3.30	4.28	+0.98
Mississippi	1.15	2.32	+1.17
Texas	9.33	11.13	+1.80

Tbl turns a simple description of a table into a **troff** or **nroff** program (list of commands) that prints the table. **Tbl** attempts to isolate a portion of a job that it can handle, then leaves the remainder for other programs. Thus **tbl** may be used with the equation formatting program **eqn** and the various macro packages without interfering with their functions.

This chapter is divided into two parts. First, we give the rules for preparing **tbl** input; then we show some examples. The description of rules is precise but technical, and the beginning user may prefer to read the examples first, as they show some common table arrangements. A section explaining how to invoke **tbl** precedes the examples.

7.2 INPUT

The input to **tbl** is text for a document, with tables preceded by a **".TS"** (table start) command and followed by a **".TE"** (table end) command. **Tbl** processes the tables, generating **troff** formatting commands, and leaves the remainder of the text unchanged. The **".TS"** and **".TE"** lines are copied, too, so that **troff** macro

packages (e.g., **-mm** and **-ms**) can use these lines to delimit and place tables as they see fit. In particular, any arguments on the “.TS” or “.TE” lines are copied but otherwise ignored, and may be used to do things like change the font or invoke a keep.

The format of the input is as follows.

```
text
.TS
table
.TE
text
. . .
```

where the format of each table is as follows.

```
.TS
options ;
format .
data
.TE
```

Each table is independent, and must contain formatting information followed by the data to be entered in the table. The formatting information, which describes the individual columns and rows of the table, may be preceded by a few options that affect the entire table. A detailed description of tables is given in the next section.

As indicated above, a table contains, first, global options, then a format section describing the layout of the table entries, and then the data to be printed. The format and data are always required, but not the options. The various parts of the table are entered as follows.

7.3 GLOBAL OPTIONS

There may be a single line of options affecting the whole table. If present, this line must follow the .TS line immediately and must contain a list of option names separated by spaces, tabs, or commas, and must be terminated by a semicolon. The allowable options are:

center	center the table (default is left-adjust);
expand	make the table as wide as the current line length;
box	enclose the table in a box;
allbox	enclose each item in the table in a box;
doublebox	enclose the table in two boxes;
tab (<i>x</i>)	use <i>x</i> instead of tab to separate data items.

linesize (*n*) set lines or rules (e.g. from **box**) in *n* point type;

delim (*xy*) recognize *x* and *y* as the *eqn* delimiters.

The **tbl** program tries to keep boxed tables on one page by issuing appropriate “need” (*.ne*) commands. These requests are calculated from the number of lines in the tables, and if there are spacing commands embedded in the input, these requests may be inaccurate; use normal **troff** procedures, such as keep-release macros, in that case. If you must have a multi-page boxed table, use macros designed for this purpose, as explained in the “USAGE” section.

7.4 FORMAT KEY-LETTERS

The format section of the table specifies the layout of the columns. Each line in this section corresponds to one line of the table (except that the last line corresponds to all following lines up to the next *.T.*, if any — see below), and each line contains a key-letter for each column of the table. It is good practice to separate the key letters for each column by spaces or tabs. Key-letters are listed below.

Note: **Tbl** will accept a key letter of either case. For example, you may specify a right-adjusted column using **R** or **r**.

L indicates a left-adjusted column entry;

R indicates a right-adjusted column entry;

C indicates a centered column entry;

N indicates a numerical column entry, to be aligned with other numerical entries so that the units digits of numbers line up;

A indicates an alphabetic subcolumn; all corresponding entries are aligned on the left, and positioned so that the widest is centered within the column;

S indicates a spanned heading (i.e., the entry from the previous column continues across this column (This is not allowed for the first column.);

^ (caret, or up-arrow) indicates a vertically spanned heading (i.e., the entry from the previous row continues down through this row. (This is not allowed for the first row of the table.)

When numerical alignment is specified, a location for the decimal point is sought. The rightmost dot (.) adjacent to a digit is used as a decimal point; if there is no dot adjoining a digit, the rightmost digit is used as a units digit; if no alignment is indicated, the item is centered in the column. However, the special non-printing character string `\.` may be used to override dots and digits, or to align

alphabetic data; this string lines up where a dot normally would, and then disappears from the final output. In the example below, the items shown at the left will be aligned (in a numerical column) as shown on the right:

13	13
4.2	4.2
26.4.12	26.4.12
abc	abc
abc\.	abc
43\3.22	433.22
749.12	749.12

Note: If numerical data are used in the same column with wider **L** or **R** type table entries, the widest *number* is centered relative to the wider **L** or **R** items (**L** is used instead of **I** for readability; they have the same meaning as key-letters). Alignment within the numerical items is preserved. This is similar to the behavior of **A** type data, as explained above. However, alphabetic sub-columns (requested by the **a** key-letter) are always slightly indented relative to **L** items; if necessary, the column width is increased to force this. This is not true for **n** type entries. The **n** and **a** items should **not** be used in the same column.

For readability, the key-letters describing each column should be separated by spaces. The end of the format section is indicated by a period. The layout of the key-letters in the format section resembles the layout of the actual data in the table. Thus a simple format might appear as:

```

c s s
l n n .

```

which specifies a table of three columns. The first line of the table contains a heading centered across all three columns; each remaining line contains a left-adjusted item in the first column followed by two columns of numerical data. A sample table in this format might be:

	Overall title	
Item-a	34.22	9.1
Item-b	12.65	.02
Items: c,d,e	23	5.8
Total	69.87	14.92

Additional features of the key-letter system are detailed below.

7.4.1 Horizontal Lines

A key-letter may be replaced by ‘_’ (underscore) to indicate a horizontal line in place of the corresponding column entry, or by ‘=’ to indicate a double horizontal line. If an adjacent column contains a horizontal line, or if there are vertical lines adjoining this column, this horizontal line is extended to meet the nearby lines. If any data entry is provided for this column, it is ignored and a warning message is printed.

7.4.2 Vertical Lines

A vertical bar may be placed between column key-letters. This will cause a vertical line between the corresponding columns of the table. A vertical bar to the left of the first key-letter or to the right of the last one produces a line at the edge of the table. If two vertical bars appear between key-letters, a double vertical line is drawn.

7.4.3 Space Between Columns

A number may follow the key-letter. This indicates the amount of separation between this column and the next column. The number normally specifies the separation in ens. If the “expand” option is used, then these numbers are multiplied by a constant such that the table is as wide as the current line length. The default column separation number is 3. If the separation is changed, the worst case (largest space requested) governs.

7.4.4 Vertical Spanning

Normally, vertically spanned items extending over several rows of the table are centered in their vertical range. If a key-letter is followed by **T**, any corresponding vertically spanned item will begin at the top line of its range.

7.4.5 Font Changes

A key-letter may be followed by a string containing a font name or number preceded by the letter **F**. This indicates that the corresponding column should be in a different font from the default (usually Roman) font. All font names are one or two letters; a one-letter font name should be separated from whatever follows by a space or tab. The single letters **B**, **b**, **I**, and **i** are shorter synonyms for **fB** and **fI**. Font change commands given with the table entries override these specifications.

7.4.6 Point Size Changes

A key-letter may be followed by the letter **P** and a number to indicate the point size of the corresponding table entries. The number may be a signed digit, in which case, it is taken as an increment or decrement from the current point size. If both a point size and a column separation value are given, one or more blanks must separate them.

7.4.7 Vertical Spacing Changes

A key-letter may be followed by the letter **V** and a number to indicate the vertical line spacing to be used within a multi-line corresponding table entry. The number may be a signed digit, in which case it is taken as an increment or decrement from the current vertical spacing. A column separation value must be separated by blanks or some other specification from a vertical spacing request. This request has no effect unless the corresponding table entry is a text block (see below).

7.4.8 Column Width Indication

A key-letter may be followed by the letter **W** and a width value in parentheses. This width is used as a minimum column width. If the largest element in the column is not as wide as the width value given after the **w**, the largest element is assumed to be that wide. If the largest element in the column is wider than the specified value, its width is used. The width is also used as a default line length for included text blocks. Normal **troff** units can be used to scale the width value; otherwise, the default is **ens**. If the width specification is a unitless integer the parentheses may be omitted. If the width value is changed in a column, only the last one given will be used.

7.4.9 Equal Width Columns

A key-letter may be followed by the letter **E** to indicate equal width columns. All columns whose key-letters are followed by **e** or **E** are made the same width. This lets you obtain a group of regularly-spaced columns. Note that order of the above features is immaterial; they need not be separated by spaces, except as indicated above to avoid ambiguities involving point size and font changes. Thus a numerical column entry in italic font and 12 point type with a minimum width of 2.5 inches and separated by 6 **ens** from the next column could be specified as

```
np12w(2.5i)f I 6
```

7.4.10 Alternative Notation

Instead of listing the format of successive lines of a table on consecutive lines of the format section, successive line formats may be given on the same line, separated by commas, so that the format for the example above might have been written:

```
c s s, l n n .
```

7.4.11 Defaults

Column descriptors missing from the end of a format line are assumed to be **L**. The longest line in the format section, however, defines the number of columns in the table; extra columns in the data are ignored silently.

7.5 DATA

After you have specified the format, you may specify the data to be placed in the table. Normally, each table line is typed as one line of data. Very long input lines can be broken: any line whose last character is \ is combined with the following line. (The \ vanishes.) The data for different columns (the table entries) must be separated by tabs, or by another character specified in the *tabs* option.

7.5.1 Troff Requests Within Tables

An input line beginning with a '.' followed by anything but a number is assumed to be a **troff** request and is passed through unchanged, retaining its position in the table. So, for example, space within a table may be produced by ".sp" commands in the data.

7.5.2 Full Width Horizontal Lines

An input *line* containing an underscore (_) or an equal sign (=) will generate a single or double line, respectively, extending the full width of the *table*.

7.5.3 Single Column Horizontal Lines

An input table *entry* containing an underscore (_) or an equal sign (=) will generate a single or double line, respectively, extending the full width of the *column*. Such lines are extended to meet horizontal or vertical lines adjoining this column. To obtain either an underscore or an equal sign in a column, either precede the characters with \. or follow them with a space before the usual tab or newline.

7.5.4 Short Horizontal Lines

An input table *entry* containing only an underscore (_) generates a single line as wide as the contents of the column. It is not extended to meet adjoining lines.

7.5.5 Vertically Spanned Items

An input table entry containing only the character string \^ indicates that the table entry immediately above spans downward over this row. It is equivalent to a table format key-letter of '^'.

7.5.6 Text Blocks

In order to include a block of text as a table entry, precede it by **T{** and follow it by **T}**. Thus, the sequence

```
T{
  block of
  text
T}
```

is the way to enter, as a single entry in the table, something that

cannot conveniently be typed as a simple string between tabs.

Note: The `T}` end delimiter must begin a line; additional columns of data may follow after a tab on the same line.

If more than twenty or thirty text blocks are used in a table, various limits in the `troff` program are likely to be exceeded, producing diagnostics such as “too many string/macro names” or “too many number registers.” Text blocks are pulled out from the table, processed separately by `troff`, and replaced in the table as a solid block. If no line length is specified in the *block of text* itself, or in the table format, the default is to use

$$L * C / (N+1)$$

where L is the current line length, C is the number of table columns spanned by the text, and N is the total number of columns in the table. The other parameters (point size, font, etc.) used in setting the block of text are those in effect at the beginning of the table (including the effect of the “.TS” macro) and any table format specifications of size, spacing and font, using the `p`, `v` and `f` modifiers to the column key-letters. Commands within the text block itself are also recognized, of course. However, `troff` commands within the table data but not within the text block do not affect that block.

Note: Although any number of lines may be present in a table, only the first 200 lines are used in calculating the widths of the various columns. A multi-page table, of course, may be arranged as several single-page tables if this proves to be a problem. Other difficulties with formatting may arise because, in the calculation of column widths all table entries are assumed to be in the font and size being used when the “.TS” command was encountered, except for font and size changes indicated (a) in the table format section and (b) within the table data (as in the entry `\s+3\fl\data\fp\s0`). Therefore, although arbitrary `troff` requests may be sprinkled in a table, care must be taken to avoid confusing the width calculations; use requests such as ‘.ps’ with care.

7.6 ADDITIONAL COMMAND LINES

If the format of a table must be changed after many similar lines, as with sub-headings or summaries, the “.T.” (table continue) command can be used to change column parameters. The outline of such a table input is:

```
.TS
options ;
format .
data
...
.T.
format .
data
.T.
format .
data
.TE
```

Using this procedure, each table line can be close to its corresponding format line.

Note: It is not possible to change the number of columns, the space between columns, the global options such as *box*, or the selection of columns to be made equal width.

7.7 USAGE

To run **tbl** as a preprocessor, use the command format below.

```
tbl filename(s) | troff options
```

The usage for **nroff** is similar to that for **troff**, although not all terminals can print boxed tables directly.

For the convenience of users employing line printers without adequate driving tables or post-filters, there is a special *-TX* command line option to **tbl** which produces output that does not have fractional line motions in it. The only other command line options recognized by **tbl** are *-ms* and *-mm* which are turned into commands to fetch the corresponding macro files; usually it is more convenient to place these arguments on the **troff** part of the command line, but they are accepted by **tbl** as well.

Note that when **eqn** and **tbl** are used together on the same file, **tbl** should be used first. If there are no equations within tables, either order works, but it is usually faster to run **tbl** first, since **eqn** normally produces a larger expansion of the input than **tbl**. However, if equations are placed within tables (using the *delim* mechanism), **tbl** must be run first. Otherwise, the output will be scrambled.

Users must also beware of using equations in **n**-style columns. This nearly always results in unacceptable output, since **tbl** attempts to split numerical format items into two parts, something that's not possible with equations. To avoid this, give the *delim(xx)* table option; this prevents splitting of numerical columns within the delimiters. For example, if the **eqn** delimiters are **\$\$**, giving *delim(\$\$)* a numerical column such as "1245 \$+- 16\$" will be

divided after 1245, not after 16.

Tbl limits tables to twenty columns; however, use of more than 16 numerical columns may fail because of limits in **troff**, producing the 'too many number registers' message. **Troff** number registers used by **tbl** must be avoided by the user within tables; these include two-digit names from 31 to 99, and names of the forms x , $x+$, $x|$, x , and $x-$, where x is any lower case letter. The names $^$, $-$, and $^$ are also used in certain circumstances. To conserve number register names, the **n** and **a** formats share a register; hence the restriction above that they may not be used in the same column.

For aid in writing layout macros, **tbl** defines a number register **TW** which is the table width; it is defined by the time that the **".TE"** macro is invoked and may be used in the expansion of that macro. More importantly, to assist in laying out multi-page boxed tables the macro **T** is defined to produce the bottom lines and side lines of a boxed table, and then invoked at its end. By using this macro in the page footer, you may arrange for multi-page tables to be boxed. In particular, the **-ms** macros can be used to print a multi-page boxed table with a repeated heading by giving the argument **H** to the **".TS"** macro. If the table start macro is written

```
.TS H
```

a line of the form

```
.TH
```

must be given in the table after any table heading (or at the start if none). Material up to the **".TH"** is placed at the top of each page of table; the remaining lines in the table are placed on several pages as required. Note that this is not a feature of **tbl**, but of the **-ms** macro package.

7.8 EXAMPLES

This section consists of a number of examples intended to illustrate the features of **tbl**. The symbol $\textcircled{\text{ }}$ in the input represents a tab character.

Input:

```
.TS
box;
c c c
l l l.
Language  $\textcircled{\text{ }}$ Authors  $\textcircled{\text{ }}$ Runs on
```

```
Fortran  $\textcircled{\text{ }}$ Many  $\textcircled{\text{ }}$ Almost anything
PL/1  $\textcircled{\text{ }}$ IBM  $\textcircled{\text{ }}$ 360/370
C  $\textcircled{\text{ }}$ BTL  $\textcircled{\text{ }}$ Apollo, others
BLISS  $\textcircled{\text{ }}$ Carnegie-Mellon  $\textcircled{\text{ }}$ PDP-10,11
IDS  $\textcircled{\text{ }}$ Honeywell  $\textcircled{\text{ }}$ H6000
Pascal  $\textcircled{\text{ }}$ Stanford  $\textcircled{\text{ }}$ Apollo, others
.TE
```

Input:

```
.TS
allbox;
c s s
c c c
n n n.
AT.T Common Stock
Year  $\textcircled{\text{ }}$ Price  $\textcircled{\text{ }}$ Dividend
1971  $\textcircled{\text{ }}$ 41-54  $\textcircled{\text{ }}$ $2.60
2  $\textcircled{\text{ }}$ 41-54  $\textcircled{\text{ }}$ 2.70
3  $\textcircled{\text{ }}$ 46-55  $\textcircled{\text{ }}$ 2.87
4  $\textcircled{\text{ }}$ 40-53  $\textcircled{\text{ }}$ 3.24
5  $\textcircled{\text{ }}$ 45-52  $\textcircled{\text{ }}$ 3.40
6  $\textcircled{\text{ }}$ 51-59  $\textcircled{\text{ }}$ .95*
.TE
* (first quarter only)
```

Output:

Language	Authors	Runs on
Fortran	Many	Almost anything
PL/1	IBM	360/370
C	BTL	Apollo, others
BLISS	Carnegie-Mellon	DEC
IDS	Honeywell	H6000
Pascal	Stanford	Apollo, others

Output:

AT.T Common Stock		
Year	Price	Dividend
1971	41-54	\$2.60
2	41-54	2.70
3	46-55	2.87
4	40-53	3.24
5	45-52	3.40
6	51-59	.95*

* (first quarter only)

Input:

```
.TS
box;
c s s
c | c | c
l | l | n.
Major New York Bridges
==
Bridge @Designer @Length
-
Brooklyn @J. A. Roebling @1595
Manhattan @G. Lindenthal @1470
Williamsburg @L. L. Buck @1600
-
Queensborough @Palmer . @1182
  @ Hornbostel
-
  @ @1380
Triborough @O. H. Ammann @_
  @ @383
-
Bronx Whitestone @O. H. Ammann @2300
Throgs Neck @O. H. Ammann @1800
-
George Washington @O. H. Ammann @3500
.TE
```

Input:

```
.TS
c c
np-2 | n | .
  @Stack
  @_
1 @46
  @_
2 @23
  @_
3 @15
  @_
4 @6.5
  @_
5 @2.1
  @_
.TE
```

Output:

Major New York Bridges		
Bridge	Designer	Length
Brooklyn	J. A. Roebling	1595
Manhattan	G. Lindenthal	1470
Williamsburg	L. L. Buck	1600
Queensborough	Palmer . Hornbostel	1182
Triborough	O. H. Ammann	1380
		383
Bronx Whitestone	O. H. Ammann	2300
Throgs Neck	O. H. Ammann	1800
George Washington	O. H. Ammann	3500

Output:

	Stack
1	46
2	23
3	15
4	6.5
5	2.1

Input:

```
.TS
box;
L L L
L L _
L L |LB
L L _
L L L.
january @february @march
april @may
june @july @Months
august @september
october @november @december
.TE
```

Input:

```
.TS
box;
cfB s s s.
Composition of Foods
_
.T.
c | c s s
c | c s s
c | c | c | c.
Food @Percent by Weight
\ ^ @
\ ^ @Protein @Fat @Carbo-
\ ^ @\ ^ @\ ^ @hydrate
_
.T.
l | n | n | n.
Apples @.4 @.5 @13.0
Halibut @18.4 @5.2 @. . .
Lima beans @7.5 @.8 @22.0
Milk @3.3 @4.0 @5.0
Mushrooms @3.5 @.4 @6.0
Rye bread @9.0 @.6 @52.7
.TE
```

Output:

january	february	march
april	may	Months
june	july	
august	september	
october	november	december

Output:

Composition of Foods			
Food	Percent by Weight		
	Protein	Fat	Carbo- hydrate
Apples	.4	.5	13.0
Halibut	18.4	5.2	...
Lima beans	7.5	.8	22.0
Milk	3.3	4.0	5.0
Mushrooms	3.5	.4	6.0
Rye bread	9.0	.6	52.7

Input:

```
.TS
allbox;
cfl s s
c cw(1i) cw(1i)
lp9 lp9 lp9.
New York Area Rocks
Era @Formation @Age (years)
Precambrian @Reading Prong @>1 billion
Paleozoic @Manhattan Prong @400 million
Mesozoic @T{
.na
Newark Basin, incl.
Stockton, Lockatong, and Brunswick
formations; also Watchungs
and Palisades.
T} @200 million
Cenozoic @Coastal Plain @T{
On Long Island 30,000 years;
Cretaceous sediments redeposited
by recent glaciation.
.ad
T}
.TE
```

Output:

<i>New York Area Rocks</i>		
Era	Formation	Age (years)
Precambrian	Reading Prong	>1 billion
Paleozoic	Manhattan Prong	400 million
Mesozoic	Newark Basin, incl. Stockton, Lockatong, and Brunswick forma- tions; also Watchungs and Palisades.	200 million
Cenozoic	Coastal Plain	On Long Island 30,000 years; Cre- taceous sediments redeposited by recent glaciation.

Input:

```
.TS
box, tab( : );
cb s s s s
cp-2 s s s s
c | | c | c | c | c
c | | c | c | c | c
r2 | | n2 | n2 | n2 | n.
Readability of Text
Line Width and Leading for 10-Point Type
=
Line : Set : 1-Point : 2-Point : 4-Point
Width : Solid : Leading : Leading : Leading
-
9 Pica : \-9.3 : \-6.0 : \-5.3 : \-7.1
14 Pica : \-4.5 : \-0.6 : \-0.3 : \-1.7
19 Pica : \-5.0 : \-5.1 : 0.0 : \-2.0
31 Pica : \-3.7 : \-3.8 : \-2.4 : \-3.6
43 Pica : \-9.1 : \-9.0 : \-5.9 : \-8.8
.TE
```

Output:

Readability of Text				
Line Width and Leading for 10-Point Type				
Line Width	Set Solid	1-Point Leading	2-Point Leading	4-Point Leading
9 Pica	-9.3	-6.0	-5.3	-7.1
14 Pica	-4.5	-0.6	-0.3	-1.7
19 Pica	-5.0	-5.1	0.0	-2.0
31 Pica	-3.7	-3.8	-2.4	-3.6
43 Pica	-9.1	-9.0	-5.9	-8.8

Input:

.TS
 c s
 cip-2 s
 l n
 a n .
 Some London Transport Statistics
 (Year 1964)
 Railway route miles ⊕244
 Tube ⊕66
 Sub-surface ⊕22
 Surface ⊕156
 .sp .5
 .T.
 l r
 a r .
 Passenger traffic \- railway
 Journeys ⊕674 million
 Average length ⊕4.55 miles
 Passenger miles ⊕3,066 million
 .T.
 l r
 a r .
 Passenger traffic \- road
 Journeys ⊕2,252 million
 Average length ⊕2.26 miles
 Passenger miles ⊕5,094 million
 .T.
 l n
 a n .
 .sp .5
 Vehicles ⊕12,521
 Railway motor cars ⊕2,905
 Railway trailer cars ⊕1,269
 Total railway ⊕4,174
 Omnibuses ⊕8,347
 .T.
 l n
 a n .
 .sp .5
 Staff ⊕73,739
 Administrative, etc. ⊕5,582
 Civil engineering ⊕5,134
 Electrical eng. ⊕1,714
 Mech. eng. \- railway ⊕4,310
 Mech. eng. \- road ⊕9,152
 Railway operations ⊕8,930
 Road operations ⊕35,946
 Other ⊕2,971
 .TE

Output:

Some London Transport Statistics
 (Year 1964)

Railway route miles	244
Tube	66
Sub-surface	22
Surface	156
Passenger traffic - railway	
Journeys	674 million
Average length	4.55 miles
Passenger miles	3,066 million
Passenger traffic - road	
Journeys	2,252 million
Average length	2.26 miles
Passenger miles	5,094 million
Vehicles	12,521
Railway motor cars	2,905
Railway trailer cars	1,269
Total railway	4,174
Omnibuses	8,347
Staff	73,739
Administrative, etc.	5,582
Civil engineering	5,134
Electrical eng.	1,714
Mech. eng. - railway	4,310
Mech. eng. - road	9,152
Railway operations	8,930
Road operations	35,946
Other	2,971

Input:

.ps 8
 .vs 10p
 .TS
 center box;
 c s s
 ci s s
 c c c
 IB l n.
 New Jersey Representatives
 (Democrats)
 .sp .5
 Name Ⓢ Office address Ⓢ Phone
 .sp .5
 James J. Florio Ⓢ 23 S. White Horse Pike, Somerdale 08083 Ⓢ 609-627-8222
 William J. Hughes Ⓢ 2920 Atlantic Ave., Atlantic City 08401 Ⓢ 609-345-4844
 James J. Howard Ⓢ 801 Bangs Ave., Asbury Park 07712 Ⓢ 201-774-1600
 Frank Thompson, Jr. Ⓢ 10 Rutgers Pl., Trenton 08618 Ⓢ 609-599-1619
 Andrew Maguire Ⓢ 115 W. Passaic St., Rochelle Park 07662 Ⓢ 201-843-0240
 Robert A. Roe Ⓢ U.S.P.O., 194 Ward St., Paterson 07510 Ⓢ 201-523-5152
 Henry Helstoski Ⓢ 666 Paterson Ave., East Rutherford 07073 Ⓢ 201-939-9090
 Peter W. Rodino, Jr. Ⓢ Suite 1435A, 970 Broad St., Newark 07102 Ⓢ 201-645-3213
 Joseph G. Minish Ⓢ 308 Main St., Orange 07050 Ⓢ 201-645-6363
 Helen S. Meyner Ⓢ 32 Bridge St., Lambertville 08530 Ⓢ 609-397-1830
 Dominick V. Daniels Ⓢ 895 Bergen Ave., Jersey City 07306 Ⓢ 201-659-7700
 Edward J. Patten Ⓢ Natl. Bank Bldg., Perth Amboy 08861 Ⓢ 201-826-4610
 .sp .5
 .T.
 ci s s
 IB l n.
 (Republicans)
 .sp .5v
 Millicent Fenwick Ⓢ 41 N. Bridge St., Somerville 08876 Ⓢ 201-722-8200
 Edwin B. Forsythe Ⓢ 301 Mill St., Moorestown 08057 Ⓢ 609-235-6622
 Matthew J. Rinaldo Ⓢ 1961 Morris Ave., Union 07083 Ⓢ 201-687-4235
 .TE
 .ps 10
 .vs 12p

Output:

New Jersey Representatives (<i>Democrats</i>)		
Name	Office address	Phone
James J. Florio	23 S. White Horse Pike, Somerdale 08083	609-627-8222
William J. Hughes	2920 Atlantic Ave., Atlantic City 08401	609-345-4844
James J. Howard	801 Bangs Ave., Asbury Park 07712	201-774-1600
Frank Thompson, Jr.	10 Rutgers Pl., Trenton 08618	609-599-1619
Andrew Maguire	115 W. Passaic St., Rochelle Park 07662	201-843-0240
Robert A. Roe	U.S.P.O., 194 Ward St., Paterson 07510	201-523-5152
Henry Helstoski	666 Paterson Ave., East Rutherford 07073	201-939-9090
Peter W. Rodino, Jr.	Suite 1435A, 970 Broad St., Newark 07102	201-645-3213
Joseph G. Minish	308 Main St., Orange 07050	201-645-6363
Helen S. Meyner	32 Bridge St., Lambertville 08530	609-397-1830
Dominick V. Daniels	895 Bergen Ave., Jersey City 07306	201-659-7700
Edward J. Patten	Natl. Bank Bldg., Perth Amboy 08861	201-826-4610
<i>(Republicans)</i>		
Millicent Fenwick	41 N. Bridge St., Somerville 08876	201-722-8200
Edwin B. Forsythe	301 Mill St., Moorestown 08057	609-235-6622
Matthew J. Rinaldo	1961 Morris Ave., Union 07083	201-687-4235

This is a paragraph of normal text placed here only to indicate where the left and right margins are. In this way, the reader can judge the appearance of centered tables or expanded tables, and observe how such tables are formatted.

Input:

```
.TS
expand;
c s s s
c c c c
l l n n.
Bell Labs Locations
Name @Address @Area Code @Phone
Holmdel @Holmdel, N. J. 07733 @201 @949-3000
Murray Hill @Murray Hill, N. J. 07974 @201 @582-6377
Whippany @Whippany, N. J. 07981 @201 @386-3000
Indian Hill @Naperville, Illinois 60540 @312 @690-2000
.TE
```

Output:

Bell Labs Locations			
Name	Address	Area Code	Phone
Holmdel	Holmdel, N. J. 07733	201	949-3000
Murray Hill	Murray Hill, N. J. 07974	201	582-6377
Whippany	Whippany, N. J. 07981	201	386-3000
Indian Hill	Naperville, Illinois 60540	312	690-2000

Input:

```
.TS
box;
cb s s s
c | c | c s
ltiw(1i) | ltw(2i) | lp8 | lw(1.6i)p8.
Some Interesting Places
```

```

Name ⊕Description ⊕Practical Information
```

```

T{
American Museum of Natural History
T} ⊕T{
The collections fill 11.5 acres (Michelin) or 25 acres (MTA)
of exhibition halls on four floors. There is a full-sized replica
of a blue whale and the world's largest star sapphire (stolen in 1964).
T} ⊕Hours ⊕10-5, ex. Sun 11-5, Wed. to 9
\^ ⊕\^ ⊕Location ⊕T{
Central Park West . 79th St.
T}
\^ ⊕\^ ⊕Admission ⊕Donation: $1.00 asked
\^ ⊕\^ ⊕Subway ⊕AA to 81st St.
\^ ⊕\^ ⊕Telephone ⊕212-873-4225
```

```

Bronx Zoo ⊕T{
About a mile long and .6 mile wide, this is the largest zoo in America.
A lion eats 18 pounds
of meat a day while a sea lion eats 15 pounds of fish.
T} ⊕Hours ⊕T{
10-4:30 winter, to 5:00 summer
T}
\^ ⊕\^ ⊕Location ⊕T{
185th St. . Southern Blvd, the Bronx.
T}
\^ ⊕\^ ⊕Admission ⊕$1.00, but Tu,We,Th free
\^ ⊕\^ ⊕Subway ⊕2, 5 to East Tremont Ave.
\^ ⊕\^ ⊕Telephone ⊕212-933-1759
```

```

Brooklyn Museum ⊕T{
Five floors of galleries contain American and ancient art.
There are American period rooms and architectural ornaments saved
```

from wreckers, such as a classical figure from Pennsylvania Station.

T} ⊕Hours ⊕Wed-Sat, 10-5, Sun 12-5

\^ ⊕\^ ⊕Location ⊕T{

Eastern Parkway . Washington Ave., Brooklyn .

T}

\^ ⊕\^ ⊕Admission ⊕Free

\^ ⊕\^ ⊕Subway ⊕2,3 to Eastern Parkway .

\^ ⊕\^ ⊕Telephone ⊕212-638-5000

⊖

T{
New York Historical Society

T} ⊕T{

All the original paintings for Audubon's

.I

Birds of America

.R

are here, as are exhibits of American decorative arts, New York history,
Hudson River school paintings, carriages, and glass paperweights.

T} ⊕Hours ⊕T{

Tues-Fri . Sun, 1-5; Sat 10-5

T}

\^ ⊕\^ ⊕Location ⊕T{

Central Park West . 77th St .

T}

\^ ⊕\^ ⊕Admission ⊕Free

\^ ⊕\^ ⊕Subway ⊕AA to 81st St .

\^ ⊕\^ ⊕Telephone ⊕212-873-3400

.TE

Output:

Some Interesting Places			
Name	Description	Practical Information	
<i>American Museum of Natural History</i>	The collections fill 11.5 acres (Michelin) or 25 acres (MTA) of exhibition halls on four floors. There is a full-sized replica of a blue whale and the world's largest star sapphire (stolen in 1964).	Hours Location Admission Subway Telephone	10-5, ex. Sun 11-5, Wed. to 9 Central Park West . 79th St. Donation: \$1.00 asked AA to 81st St. 212-873-4225
<i>Bronx Zoo</i>	About a mile long and .6 mile wide, this is the largest zoo in America. A lion eats 18 pounds of meat a day while a sea lion eats 15 pounds of fish.	Hours Location Admission Subway Telephone	10-4:30 winter, to 5:00 summer 185th St. . Southern Blvd, the Bronx. \$1.00, but Tu, We, Th free 2, 5 to East Tremont Ave. 212-933-1759
<i>Brooklyn Museum</i>	Five floors of galleries contain American and ancient art. There are American period rooms and architectural ornaments saved from wreckers, such as a classical figure from Pennsylvania Station.	Hours Location Admission Subway Telephone	Wed-Sat, 10-5, Sun 12-5 Eastern Parkway . Washington Ave., Brooklyn. Free 2,3 to Eastern Parkway. 212-638-5000
<i>New York Historical Society</i>	All the original paintings for Audubon's <i>Birds of America</i> are here, as are exhibits of American decorative arts, New York history, Hudson River school paintings, carriages, and glass paperweights.	Hours Location Admission Subway Telephone	Tues-Fri . Sun, 1-5; Sat 10-5 Central Park West . 77th St. Free AA to 81st St. 212-873-3400

7.9 SUMMARY OF COMMANDS AND KEY-LETTERS

The following table summarizes all of **tbl**'s commands and key-letters.

Note: All key-letters are acceptable in either upper- or lower-case. Only upper case is shown in the table below.

<i>Command</i>	<i>Meaning</i>
A	Alphabetic subcolumn
allbox	Draw box around all items
B	Boldface item
box	Draw box around table
C	Centered column
center	Center table in page
doublebox	Doubled box around table
E	Equal width columns
expand	Make table full line width
F	Font change
I	Italic item
L	Left adjusted column
N	Numerical column
nnn	Column separation
P	Point size change
R	Right adjusted column
S	Spanned item
T	Vertical spanning at top
tab (x)	Change data separator character
T{" ~ "T}"	Text block
V	Vertical spacing change
W	Minimum width value
.xx	Included <i>troff</i> command
	Vertical line
	Double vertical line
^	Vertical span
\^	Vertical span
=	Double horizontal line
—	Horizontal line
⌋	Short horizontal line



Index

!, in troff conditional request	2-30	footer, and header in -mm	5-7
;, as tbl delimiter	7-2	footnote	
{, as troff delimiter	2-30	in me	4-7
		with equation	6-8
A			
accent marks		H	
in eqn	6-8	header, and footer in -mm	5-7
-ms "new"	3-10	headings	
arithmetic operators, in troff	1-17, 2-5	in -mm	5-14, 5-16
		numbering style in -mm	5-17
B		heads, numbered, in -me	4-3
backslash, as troff escape	1-2	hyphenation, in n/troff	2-10
beginning, of -mm document	5-2	hyphenation indicator, in -mm	5-11
blank line, in troff	2-14		
body, of -mm document	5-2	I	
brace, as eqn delimiter	6-6	indent, hanging	
bullet, in -mm list	5-24	in -me	4-2
		in -ms	3-4
C		indents, nested, in -ms	3-5
characters		index, in -me	4-7
troff greek	1-6		
troff special	1-6	L	
col		leading, in troff	1-3
used with nroff and -mm	5-6	line, horizontal, in tbl	7-4
used with nroff files	3-12	line, vertical, in tbl	7-5
column width, in tbl	7-6	line overflow, n/troff error	2-33, 6-18
comment, in n/troff	1-14, 2-24	lists, nested, in -mm	5-21
compact macros, -mm	5-4		
		M	
D		me macros,	
debug mode, in -mm	5-6	\$0	4-4
decimal point, in tables	7-3	\$C	4-12
diacritical marks, in troff	1-11	\$c	4-12
display queue, in -mm	5-33	\$p	4-4
dollar sign, as eqn delimiter	6-12	(c	4-6
		(f	4-7
E		(l	4-6
em, in nroff	2-4	(q	4-6
end, of -mm document	5-2	(x	4-7
equations, in -me	4-10	(z	4-6
)c	4-7
F)q	4-6
font)z	4-6
in tables	7-5	+c	4-12
to set in in -me	4-1	2c	4-8
to change in -ms	3-6	ac	4-12

B	4-6	HM	5-17
bc	4-8	HU	5-18
bi	4-9	HX	5-19
bx	4-9	HY	5-19
ef	4-5	HZ	5-19
eh	4-5	I	5-49
fo	4-4	LB	5-27
he	4-4	LB	5-29
hl	4-11	LE	5-24
hx	4-5	LI	5-23
i	4-8	ML	5-25
ip	4-2	ND	5-55
ix	4-9	NS	5-56
lo	4-11	OF	5-42
lp	4-2	OP	5-53
n1	4-9	P	5-13
n2	4-9	PF	5-42
of	4-5	PF	5-42
oh	4-5	PH	5-41
pd	4-7	PS	5-53
pp	4-2	PX	5-44
R	4-6	R	5-49
r	4-8	RD	5-54
rb	4-8	RE	5-50
re	4-10	RF	5-48
sh	4-3	RL	5-26
sk	4-9	RS	5-48
sx	4-3	SA	5-50
sz	4-8	SK	5-53
TE	7-1	SP	5-52
th	4-11	TB	5-37
tp	4-11	TC	5-45
TS	7-1	TE	7-1
u	4-9	TH	5-36
uh	4-3	TP	5-43
mm macros,		TS	7-1
for BTL use	5-54	TX	5-46
2C	5-50	TY	5-46
AL	5-24	VL	5-26
AV	5-57	VM	5-45
B	5-49	WC	5-35
BE	5-44	ms macros,	
BL	5-25	2C	3-2
DF	5-33	AM	3-10
DL	5-25	B1	3-8
DS	5-32	B2	3-8
EF	5-42	BX	3-8
EX	5-37	DA	3-9
FD	5-39	DE	3-8
FE	5-38	DS	3-8
FG	5-37	EQ	6-2
FS	5-38	FE	3-7
H	5-14	FS	3-7

I	3-6	semicolon, as tbl delimiter	7-2
IP	3-4	spaces	
KF	3-9	in tables	7-5
KS	3-9	interpreted by eqn	6-2, 6-3
LP	3-2	special characters	
ND	3-9	as troff delimiters	2-21
NH	3-3	in -mm	5-11
PB	3-2	in troff	2-5
PP	3-2	string, to define in troff	1-12
PT	3-2		
PX	3-13		
R	3-6	T	
RE	3-5	tab stops, in n/troff	2-20
RS	3-5	table	
TE	7-1	example of simple format	7-4
TS	7-1	in -me	4-10
XE	3-13	tabs	
XP	3-12	in -me	4-10
XS	3-13	in -mm	5-11
		in troff	1-8
		text filling, in troff	2-11
		tilde	
		as eqn delimiter	6-3
		in n/troff	2-4
		trap, in troff	1-15
		troff, command line options	2-1
		troff escape character, 2-3	
		troff font change, in-line	1-5
		troff requests	
		ad	2-11
		am	2-17
		as	2-18
		bd	2-8
		bp	2-9
		c2	2-23
		cc	2-23
		ce	2-12
		ce	2-14
		ch	2-18
		cs	2-7
		cu	2-23
		cu	5-16
		da	2-18
		de	1-13
		de	2-15
		de	2-17
		di	1-23
		di	2-18
		ds	1-12
		ds	2-17
		ec	2-22
		em	2-19
		eo	2-22
		ev	1-22
N			
newlines, interpreted by eqn	6-2		
number registers			
to set, in -mm	5-6		
used by -ms	3-9		
P			
page number			
in -ms	3-2		
in troff	1-16		
paragraph			
in -me	4-2		
in -mm	5-13		
in -ms	3-2		
pipes, in troff command lines	2-3		
point size			
in -mm headings	5-16		
in tables	7-5		
in troff	1-2, 2-6		
relative change	1-3		
to change in -ms	3-6		
preamble, of -mm document	5-1		
Q			
quote			
in eqn	6-9		
in -mm macro arguments	5-9		
R			
resolution, in n/troff	1-4, 2-4		
S			
scale indicators, in n/troff	2-4		

ex	2-31	sp	2-13
fc	2-21	ss	2-7
fi	2-11	sv	2-13
fl	2-33	sv	2-13
fp	1-5	ta	2-20
fp	2-6	ta	2-21
fp	2-8	tc	1-8
ft	1-5	tc	2-21
ft	2-6	ti	2-14
ft	2-8	tl	1-15
if	1-21	tm	2-32
if	2-30	tr	1-6
ig	2-32	tr	2-24
in	1-14	uf	2-23
in	2-14	ul	1-5
it	2-18	ul	2-23
lc	2-21	ul	5-16
lg	2-22	vs	2-13
ll	1-6	wh	1-15
ll	2-14	wh	1-15
ls	2-13	wh	2-17
lt	1-16	wh	2-18
lt	2-14		
lt	2-28	V	
mc	2-32	vertical spacing, in tables	7-5
mk	2-10		
ne	2-9	W	
nf	2-11	word overflow, n/troff error	2-33, 6-18
nm	2-29		
nn	2-29		
nr	1-17		
nr	2-19		
nr	2-19		
nr	5-8		
ns	2-14		
nx	2-32		
pc	2-28		
pi	2-32		
pi	3-12		
pl	2-9		
pm	2-33		
pn	2-9		
po	2-9		
ps	2-7		
rd	2-31		
rm	2-18		
rn	2-18		
rr	2-20		
rs	2-14		
rt	2-10		
so	2-32		
sp	1-6		
sp	2-13		

Appendix A : Advanced Editing on UNIX

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

Although UNIX provides remarkably effective tools for text editing, that by itself is no guarantee that everyone will automatically make the most effective use of them. In particular, people who are not computer specialists — typists, secretaries, casual users — often use the system less effectively than they might.

This document is intended as a sequel to *A Tutorial Introduction to the UNIX Text Editor* [1], providing explanations and examples of how to edit with less effort. (You should also be familiar with the material in *UNIX For Beginners* [2].) Further information on all commands discussed here can be found in *The UNIX Programmer's Manual* [3].

Examples are based on observations of users and the difficulties they encounter. Topics covered include special characters in searches and substitute commands, line addressing, the global commands, and line moving and copying. There are also brief discussions of effective use of related tools, like those for file manipulation, and those based on `ed`, like `grep` and `sed`.

A word of caution. There is only one way to learn to use something, and that is to *use* it. Reading a description is no substitute for trying something. A paper like this one should give you ideas about what to try, but until you actually try something, you will not learn it.

2. SPECIAL CHARACTERS

The editor `ed` is the primary interface to the system for many people, so it is worthwhile to know how to get the most out of `ed` for the least effort.

The next few sections will discuss shortcuts and labor-saving devices. Not all of these will be instantly useful to any one person, of course, but a few will be, and the others should give you ideas to store away for future use. And as always, until you try these things, they will remain theoretical knowledge, not something you have confidence in.

The List command 'l'

`ed` provides two commands for printing the contents of the lines you're editing. Most people are familiar with `p`, in combinations like

1,\$p

to print all the lines you're editing, or

s/abc/def/p

to change 'abc' to 'def' on the current line. Less familiar is the *list* command **l** (the letter '*l*'), which gives slightly more information than **p**. In particular, **l** makes visible characters that are normally invisible, such as tabs and backspaces. If you list a line that contains some of these, **l** will print each tab as \gt and each backspace as \lt . This makes it much easier to correct the sort of typing mistake that inserts extra spaces adjacent to tabs, or inserts a backspace followed by a space.

The **l** command also 'folds' long lines for printing — any line that exceeds 72 characters is printed on multiple lines; each printed line except the last is terminated by a backslash \backslash , so you can tell it was folded. This is useful for printing long lines on short terminals.

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

The Substitute Command 's'

Most of the next few sections will be taken up with a discussion of the substitute command **s**. Since this is the command for changing the contents of individual lines, it probably has the most complexity of any **ed** command, and the most potential for effective use.

As the simplest place to begin, recall the meaning of a trailing **g** after a substitute command. With

s/this/that/

and

s/this/that/g

the first one replaces the *first* 'this' on the line with 'that'. If there is more than one 'this' on the line, the second form with the trailing **g** changes *all* of them.

Either form of the **s** command can be followed by **p** or **l** to 'print' or 'list' (as described in the previous section) the contents of the line:

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

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

Of course, any **s** command can be preceded by one or two 'line numbers' to specify that the substitution is to take place on a group of lines. Thus


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

changes the *first* occurrence of 'mispell' to 'misspell' on every line of the file. But

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

changes *every* occurrence in every line (and this is more likely to be what you wanted in this particular case).

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

The Undo Command 'u'

Occasionally you will make a substitution in a line, only to realize too late that it was a ghastly mistake. The 'undo' command **u** lets you 'undo' the last substitution: the last line that was substituted can be restored to its previous state by typing the command

```
u
```

The Metacharacter '.'

As you have undoubtedly noticed when you use **ed**, certain characters have unexpected meanings when they occur in the left side of a substitute command, or in a search for a particular line. In the next several sections, we will talk about these special characters, which are often called 'metacharacters'.

The first one is the period '.'. On the left side of a substitute command, or in a search with '/.../', '.' stands for *any* single character. Thus the search

```
/x.y/
```

finds any line where 'x' and 'y' occur separated by a single character, as in

```
x+y
```

```
x-y
```

```
x y
```

```
x.y
```

and so on. (We will use to stand for a space whenever we need to make it visible.)

Since '.' matches a single character, that gives you a way to deal with funny characters printed by **l**. Suppose you have a line that, when printed with the **l** command, appears as

```
.... th\07is ....
```

and you want to get rid of the `\07` (which represents the bell character, by the way).

The most obvious solution is to try

```
s/\07//
```

but this will fail. (Try it.) The brute force solution, which most people would now take, is to re-type the entire line. This is guaranteed, and is actually quite a reasonable tactic if the line in question isn't too big, but for a very long line, re-

typing is a bore. This is where the metacharacter '.' comes in handy. Since '\07' really represents a single character, if we say

```
s/th.is/this/
```

the job is done. The '.' matches the mysterious character between the 'h' and the 'i', *whatever it is*.

Bear in mind that since '.' matches any single character, the command

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

converts the first character on a line into a ',', which very often is not what you intended.

As is true of many characters in **ed**, the '.' has several meanings, depending on its context. This line shows all three:

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

The first '.' is a line number, the number of the line we are editing, which is called 'line dot'. (We will discuss line dot more in Section 3.) The second '.' is a metacharacter that matches any single character on that line. The third '.' is the only one that really is an honest literal period. On the *right* side of a substitution, '.' is not special. If you apply this command to the line

```
Now is the time.
```

the result will be

```
.ow is the time.
```

which is probably not what you intended.

The Backslash '\'

Since a period means 'any character', the question naturally arises of what to do when you really want a period. For example, how do you convert the line

```
Now is the time.
```

into

```
Now is the time?
```

The backslash '\' does the job. A backslash turns off any special meaning that the next character might have; in particular, '\.' converts the '.' from a 'match anything' into a period, so you can use it to replace the period in

```
Now is the time.
```

like this:

```
s/\./?/
```

The pair of characters '\.' is considered by **ed** to be a single real period.

The backslash can also be used when searching for lines that contain a special character. Suppose you are looking for a line that contains

```
.PP
```

The search

`/.PP/`

isn't adequate, for it will find a line like

THE APPLICATION OF ...

because the '.' matches the letter 'A'. But if you say

`/\ .PP/`

you will find only lines that contain '.PP'.

The backslash can also be used to turn off special meanings for characters other than '.'. For example, consider finding a line that contains a backslash. The search

`/\`

won't work, because the '\ ' isn't a literal '\ ', but instead means that the second '/' no longer delimits the search. But by preceding a backslash with another one, you can search for a literal backslash. Thus

`\\/`

does work. Similarly, you can search for a forward slash '/' with

`/\/`

The backslash turns off the meaning of the immediately following '/' so that it doesn't terminate the `/.../` construction prematurely.

As an exercise, before reading further, find two substitute commands each of which will convert the line

`\x\ .y`

into the line

`\x\y`

Here are several solutions; verify that each works as advertised.

`s/\\/ .//`

`s/x . /x/`

`s/ . y /y/`

A couple of miscellaneous notes about backslashes and special characters. First, you can use any character to delimit the pieces of an `s` command: there is nothing sacred about slashes. (But you must use slashes for context searching.) For instance, in a line that contains a lot of slashes already, like

`//exec //sys.fort.go // etc...`

you could use a colon as the delimiter — to delete all the slashes, type

`s/::g`

Second, if `#` and `@` are your character erase and line kill characters, you have to type `\#` and `\@`; this is true whether you're talking to `ed` or any other program.

When you are adding text with **a** or **i** or **c**, backslash is not special, and you should only put in one backslash for each one you really want.

The Dollar Sign '\$'

The next metacharacter, the '\$', stands for 'the end of the line'. As its most obvious use, suppose you have the line

Now is the

and you wish to add the word 'time' to the end. Use the \$ like this:

s/\$/ time/

to get

Now is the time

Notice that a space is needed before 'time' in the substitute command, or you will get

Now is thetime

As another example, replace the second comma in the following line with a period without altering the first:

Now is the time, for all good men,

The command needed is

s/,,\$/./

The \$ sign here provides context to make specific which comma we mean. Without it, of course, the s command would operate on the first comma to produce

Now is the time. for all good men,

As another example, to convert

Now is the time.

into

Now is the time?

as we did earlier, we can use

s/.\$/?/

Like '.', the '\$' has multiple meanings depending on context. In the line

\$s/\$/\$/

the first '\$' refers to the last line of the file, the second refers to the end of that line, and the third is a literal dollar sign, to be added to that line.

The Circumflex '^'

The circumflex (or hat or caret) '^' stands for the beginning of the line. For example, suppose you are looking for a line that begins with 'the'. If you simply say

```
/the/
```

you will in all likelihood find several lines that contain 'the' in the middle before arriving at the one you want. But with

```
/^the/
```

you narrow the context, and thus arrive at the desired one more easily.

The other use of '^' is of course to enable you to insert something at the beginning of a line:

```
s/^/ /
```

places a space at the beginning of the current line.

Metacharacters can be combined. To search for a line that contains *only* the characters

```
.PP
```

you can use the command

```
/^\.PP$/
```

The Star '*'

Suppose you have a line that looks like this:

```
text x          y text
```

where *text* stands for lots of text, and there are some indeterminate number of spaces between the *x* and the *y*. Suppose the job is to replace all the spaces between *x* and *y* by a single space. The line is too long to retype, and there are too many spaces to count. What now?

This is where the metacharacter '*' comes in handy. A character followed by a star stands for as many consecutive occurrences of that character as possible. To refer to all the spaces at once, say

```
s/x *y/x y/
```

The construction ' *' means 'as many spaces as possible'. Thus 'x *y' means 'an x, as many spaces as possible, then a y'.

The star can be used with any character, not just space. If the original example was instead

```
text x—————y text
```

then all '-' signs can be replaced by a single space with the command

```
s/x-*y/x y/
```

Finally, suppose that the line was

text x.....y text

Can you see what trap lies in wait for the unwary? If you blindly type

*s/x.*y/x y/*

what will happen? The answer, naturally, is that it depends. If there are no other x's or y's on the line, then everything works, but it's blind luck, not good management. Remember that '.' matches *any* single character? Then '.' matches as many single characters as possible, and unless you're careful, it can eat up a lot more of the line than you expected. If the line was, for example, like this:

text x text x.....y text y text

then saying

*s/x.*y/x y/*

will take everything from the *first* 'x' to the *last* 'y', which, in this example, is undoubtedly more than you wanted.

The solution, of course, is to turn off the special meaning of '.' with '\.':

*s/x\.*y/x y/*

Now everything works, for '\.*' means 'as many *periods* as possible'.

There are times when the pattern '\.*' is exactly what you want. For example, to change

Now is the time for all good men

into

Now is the time.

use '\.*' to eat up everything after the 'for':

s/ for../*

There are a couple of additional pitfalls associated with '*' that you should be aware of. Most notable is the fact that 'as many as possible' means *zero* or more. The fact that zero is a legitimate possibility is sometimes rather surprising. For example, if our line contained

text xy text x y text

and we said

*s/x *y/x y/*

the *first* 'xy' matches this pattern, for it consists of an 'x', zero spaces, and a 'y'. The result is that the substitute acts on the first 'xy', and does not touch the later one that actually contains some intervening spaces.

The way around this, if it matters, is to specify a pattern like

*/x *y/*

which says 'an x, a space, then as many more spaces as possible, then a y', in other words, one or more spaces.

The other startling behavior of '*' is again related to the fact that zero is a legitimate number of occurrences of something followed by a star. The command

```
s/x*/y/g
```

when applied to the line

```
abcdef
```

produces

```
yaybycydyeyfy
```

which is almost certainly not what was intended. The reason for this behavior is that zero is a legal number of matches, and there are no x's at the beginning of the line (so that gets converted into a 'y'), nor between the 'a' and the 'b' (so that gets converted into a 'y'), nor ... and so on. Make sure you really want zero matches; if not, in this case write

```
s/xx*/y/g
```

'xx*' is one or more x's.

The Brackets '[']'

Suppose that you want to delete any numbers that appear at the beginning of all lines of a file. You might first think of trying a series of commands like

```
1,$s/^1*//
```

```
1,$s/^2*//
```

```
1,$s/^3*//
```

and so on, but this is clearly going to take forever if the numbers are at all long. Unless you want to repeat the commands over and over until finally all numbers are gone, you must get all the digits on one pass. This is the purpose of the brackets [and].

The construction

```
[0123456789]
```

matches any single digit — the whole thing is called a 'character class'. With a character class, the job is easy. The pattern '[0123456789]*' matches zero or more digits (an entire number), so

```
1,$s/^[0123456789]*//
```

deletes all digits from the beginning of all lines.

Any characters can appear within a character class, and just to confuse the issue there are essentially no special characters inside the brackets; even the backslash doesn't have a special meaning. To search for special characters, for example, you can say

```
/[\.$^[]/
```

Within [...], the '[' is not special. To get a '[' into a character class, make it the first character.

It's a nuisance to have to spell out the digits, so you can abbreviate them as [0-9]; similarly, [a-z] stands for the lower case letters, and [A-Z] for upper case.

As a final frill on character classes, you can specify a class that means 'none of the following characters'. This is done by beginning the class with a '^':

```
[^0-9]
```

stands for 'any character *except* a digit'. Thus you might find the first line that doesn't begin with a tab or space by a search like

```
/^[^(space)(tab)]/
```

Within a character class, the circumflex has a special meaning only if it occurs at the beginning. Just to convince yourself, verify that

```
/^[^]/
```

finds a line that doesn't begin with a circumflex.

The Ampersand '&'

The ampersand '&' is used primarily to save typing. Suppose you have the line

```
Now is the time
```

and you want to make it

```
Now is the best time
```

Of course you can always say

```
s/the/the best/
```

but it seems silly to have to repeat the 'the'. The '&' is used to eliminate the repetition. On the *right* side of a substitute, the ampersand means 'whatever was just matched', so you can say

```
s/the/& best/
```

and the '&' will stand for 'the'. Of course this isn't much of a saving if the thing matched is just 'the', but if it is something truly long or awful, or if it is something like '.' which matches a lot of text, you can save some tedious typing. There is also much less chance of making a typing error in the replacement text. For example, to parenthesize a line, regardless of its length,

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

The ampersand can occur more than once on the right side:

```
s/the/& best and & worst/
```

makes

```
Now is the best and the worst time
```

and

```
s/.*/&? &!!/
```

converts the original line into

```
Now is the time? Now is the time!!
```


To get a literal ampersand, naturally the backslash is used to turn off the special meaning:

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

converts the word into the symbol. Notice that '&' is not special on the left side of a substitute, only on the *right* side.

Substituting Newlines

ed provides a facility for splitting a single line into two or more shorter lines by 'substituting in a newline'. As the simplest example, suppose a line has gotten unmanageably long because of editing (or merely because it was unwisely typed). If it looks like

```
text xy text
```

you can break it between the 'x' and the 'y' like this:

```
s/xy/x\  
y/
```

This is actually a single command, although it is typed on two lines. Bearing in mind that '\' turns off special meanings, it seems relatively intuitive that a '\' at the end of a line would make the newline there no longer special.

You can in fact make a single line into several lines with this same mechanism. As a large example, consider underlining the word 'very' in a long line by splitting 'very' onto a separate line, and preceding it by the **roff** or **nroff** formatting command '.ul'.

```
text a very big text
```

The command

```
s/ very /\  
.ul\  
very\  
/
```

converts the line into four shorter lines, preceding the word 'very' by the line '.ul', and eliminating the spaces around the 'very', all at the same time.

When a newline is substituted in, dot is left pointing at the last line created.

Joining Lines

Lines may also be joined together, but this is done with the **j** command instead of **s**. Given the lines

```
Now is  
the time
```

and supposing that dot is set to the first of them, then the command

```
j
```

joins them together. No blanks are added, which is why we carefully showed a blank at the beginning of the second line.

All by itself, a **j** command joins line dot to line dot+1, but any contiguous set of lines can be joined. Just specify the starting and ending line numbers. For example,

```
1,$jp
```

joins all the lines into one big one and prints it. (More on line numbers in Section 3.)

Rearranging a Line with `\(... \)`

(This section should be skipped on first reading.) Recall that `'&'` is a shorthand that stands for whatever was matched by the left side of an **s** command. In much the same way you can capture separate pieces of what was matched; the only difference is that you have to specify on the left side just what pieces you're interested in.

Suppose, for instance, that you have a file of lines that consist of names in the form

```
Smith, A. B.  
Jones, C.
```

and so on, and you want the initials to precede the name, as in

```
A. B. Smith  
C. Jones
```

It is possible to do this with a series of editing commands, but it is tedious and error-prone. (It is instructive to figure out how it is done, though.)

The alternative is to `'tag'` the pieces of the pattern (in this case, the last name, and the initials), and then rearrange the pieces. On the left side of a substitution, if part of the pattern is enclosed between `\(` and `\)`, whatever matched that part is remembered, and available for use on the right side. On the right side, the symbol `'\1'` refers to whatever matched the first `\(...\)` pair, `'\2'` to the second `\(...\)`, and so on.

The command

```
1,$s/^\([^,]*\) , *\(...\) / \2 \1/
```

although hard to read, does the job. The first `\(...\)` matches the last name, which is any string up to the comma; this is referred to on the right side with `'\1'`. The second `\(...\)` is whatever follows the comma and any spaces, and is referred to as `'\2'`.

Of course, with any editing sequence this complicated, it's foolhardy to simply run it and hope. The global commands **g** and **v** discussed in section 4 provide a way for you to print exactly those lines which were affected by the substitute command, and thus verify that it did what you wanted in all cases.

3. LINE ADDRESSING IN THE EDITOR

The next general area we will discuss is that of line addressing in **ed**, that is, how you specify what lines are to be affected by editing commands. We have already used constructions like

1,\$s/x/y/

to specify a change on all lines. And most users are long since familiar with using a single newline (or return) to print the next line, and with

/thing/

to find a line that contains 'thing'. Less familiar, surprisingly enough, is the use of

?thing?

to scan *backwards* for the previous occurrence of 'thing'. This is especially handy when you realize that the thing you want to operate on is back up the page from where you are currently editing.

The slash and question mark are the only characters you can use to delimit a context search, though you can use essentially any character in a substitute command.

Address Arithmetic

The next step is to combine the line numbers like '.', '\$', '/.../' and '?...?' with '+' and '-'. Thus

\$-1

is a command to print the next to last line of the current file (that is, one line before line '\$'). For example, to recall how far you got in a previous editing session,

\$-5,\$p

prints the last six lines. (Be sure you understand why it's six, not five.) If there aren't six, of course, you'll get an error message.

As another example,

.-3,.+3p

prints from three lines before where you are now (at line dot) to three lines after, thus giving you a bit of context. By the way, the '+' can be omitted:

.-3,.3p

is absolutely identical in meaning.

Another area in which you can save typing effort in specifying lines is to use '-' and '+' as line numbers by themselves.

—
by itself is a command to move back up one line in the file. In fact, you can string several minus signs together to move back up that many lines:

—
moves up three lines, as does '-3'. Thus

-3,+3p

is also identical to the examples above.

Since '-' is shorter than '-1', constructions like

```
-,s/bad/good/
```

are useful. This changes 'bad' to 'good' on the previous line and on the current line.

'+' and '-' can be used in combination with searches using '/.../' and '?...?', and with '\$'. The search

```
/thing/—
```

finds the line containing 'thing', and positions you two lines before it.

Repeated Searches

Suppose you ask for the search

```
/horrible thing/
```

and when the line is printed you discover that it isn't the horrible thing that you wanted, so it is necessary to repeat the search again. You don't have to re-type the search, for the construction

```
//
```

is a shorthand for 'the previous thing that was searched for', whatever it was. This can be repeated as many times as necessary. You can also go backwards:

```
??
```

searches for the same thing, but in the reverse direction.

Not only can you repeat the search, but you can use '//' as the left side of a substitute command, to mean 'the most recent pattern'.

```
/horrible thing/
.... ed prints line with 'horrible thing' ...
s//good/p
```

To go backwards and change a line, say

```
??s//good/
```

Of course, you can still use the '&' on the right hand side of a substitute to stand for whatever got matched:

```
//s//& &/p
```

finds the next occurrence of whatever you searched for last, replaces it by two copies of itself, then prints the line just to verify that it worked.

Default Line Numbers and the Value of Dot

One of the most effective ways to speed up your editing is always to know what lines will be affected by a command if you don't specify the lines it is to act on, and on what line you will be positioned (i.e., the value of dot) when a command finishes. If you can edit without specifying unnecessary line numbers, you can save a lot of typing.

As the most obvious example, if you issue a search command like

/thing/

you are left pointing at the next line that contains 'thing'. Then no address is required with commands like **s** to make a substitution on that line, or **p** to print it, or **l** to list it, or **d** to delete it, or **a** to append text after it, or **c** to change it, or **i** to insert text before it.

What happens if there was no 'thing'? Then you are left right where you were — dot is unchanged. This is also true if you were sitting on the only 'thing' when you issued the command. The same rules hold for searches that use '?...?'; the only difference is the direction in which you search.

The delete command **d** leaves dot pointing at the line that followed the last deleted line. When line '\$' gets deleted, however, dot points at the *new* line '\$'.

The line-changing commands **a**, **c** and **i** by default all affect the current line — if you give no line number with them, **a** appends text after the current line, **c** changes the current line, and **i** inserts text before the current line.

a, **c**, and **i** behave identically in one respect — when you stop appending, changing or inserting, dot points at the last line entered. This is exactly what you want for typing and editing on the fly. For example, you can say

```
a
... text ...
... botch ...      (minor error)
.
s/botch/correct/   (fix botched line)
a
... more text ...
```

without specifying any line number for the substitute command or for the second append command. Or you can say

```
a
... text ...
... horrible botch ... (major error)
.
c                      (replace entire line)
... fixed up line ...
```

You should experiment to determine what happens if you add *no* lines with **a**, **c** or **i**.

The **r** command will read a file into the text being edited, either at the end if you give no address, or after the specified line if you do. In either case, dot points at the last line read in. Remember that you can even say **Or** to read a file in at the beginning of the text. (You can also say **Oa** or **li** to start adding text at the beginning.)

The **w** command writes out the entire file. If you precede the command by one line number, that line is written, while if you precede it by two line numbers, that range of lines is written. The **w** command does *not* change dot: the current line remains the same, regardless of what lines are written. This is true even if you say something like

```
/^\.AB/,/^\.AE/w abstract
```

which involves a context search.

Since the `w` command is so easy to use, you should save what you are editing regularly as you go along just in case the system crashes, or in case you do something foolish, like clobbering what you're editing.

The least intuitive behavior, in a sense, is that of the `s` command. The rule is simple — you are left sitting on the last line that got changed. If there were no changes, then dot is unchanged.

To illustrate, suppose that there are three lines in the buffer, and you are sitting on the middle one:

```
x1
x2
x3
```

Then the command

```
-,+s/x/y/p
```

prints the third line, which is the last one changed. But if the three lines had been

```
x1
y2
y3
```

and the same command had been issued while dot pointed at the second line, then the result would be to change and print only the first line, and that is where dot would be set.

Semicolon ';'

Searches with `/.../` and `?...?` start at the current line and move forward or backward respectively until they either find the pattern or get back to the current line. Sometimes this is not what is wanted. Suppose, for example, that the buffer contains lines like this:

```
.
.
.
ab
.
.
.
bc
.
.
```

Starting at line 1, one would expect that the command

```
/a/,/b/p
```

prints all the lines from the 'ab' to the 'bc' inclusive. Actually this is not what happens. *Both* searches (for 'a' and for 'b') start from the same point, and thus they both find the line that contains 'ab'. The result is to print a single line.

Worse, if there had been a line with a 'b' in it before the 'ab' line, then the print command would be in error, since the second line number would be less than the first, and it is illegal to try to print lines in reverse order.

This is because the comma separator for line numbers doesn't set dot as each address is processed; each search starts from the same place. In **ed**, the semicolon ';' can be used just like comma, with the single difference that use of a semicolon forces dot to be set at that point as the line numbers are being evaluated. In effect, the semicolon 'moves' dot. Thus in our example above, the command

```
/a;/b/p
```

prints the range of lines from 'ab' to 'bc', because after the 'a' is found, dot is set to that line, and then 'b' is searched for, starting beyond that line.

This property is most often useful in a very simple situation. Suppose you want to find the *second* occurrence of 'thing'. You could say

```
/thing/  
//
```

but this prints the first occurrence as well as the second, and is a nuisance when you know very well that it is only the second one you're interested in. The solution is to say

```
/thing;///
```

This says to find the first occurrence of 'thing', set dot to that line, then find the second and print only that.

Closely related is searching for the second previous occurrence of something, as in

```
?something?;??
```

Printing the third or fourth or ... in either direction is left as an exercise.

Finally, bear in mind that if you want to find the first occurrence of something in a file, starting at an arbitrary place within the file, it is not sufficient to say

```
1;/thing/
```

because this fails if 'thing' occurs on line 1. But it is possible to say

```
0;/thing/
```

(one of the few places where 0 is a legal line number), for this starts the search at line 1.

Interrupting the Editor

As a final note on what dot gets set to, you should be aware that if you hit the interrupt or delete or rubout or break key while **ed** is doing a command, things are put back together again and your state is restored as much as possible to what it was before the command began. Naturally, some changes are irrevocable — if you are reading or writing a file or making substitutions or deleting lines, these will be stopped in some clean but unpredictable state in the middle (which is why it is not usually wise to stop them). Dot may or may not be

changed.

Printing is more clear cut. Dot is not changed until the printing is done. Thus if you print until you see an interesting line, then hit delete, you are *not* sitting on that line or even near it. Dot is left where it was when the **p** command was started.

4. GLOBAL COMMANDS

The global commands **g** and **v** are used to perform one or more editing commands on all lines that either contain (**g**) or don't contain (**v**) a specified pattern.

As the simplest example, the command

```
g/UNIX/p
```

prints all lines that contain the word 'UNIX'. The pattern that goes between the slashes can be anything that could be used in a line search or in a substitute command; exactly the same rules and limitations apply.

As another example, then,

```
g/^\./p
```

prints all the formatting commands in a file (lines that begin with '.').

The **v** command is identical to **g**, except that it operates on those line that do *not* contain an occurrence of the pattern. (Don't look too hard for mnemonic significance to the letter 'v'.) So

```
v/^\./p
```

prints all the lines that don't begin with '.' — the actual text lines.

The command that follows **g** or **v** can be anything:

```
g/^\./d
```

deletes all lines that begin with '.', and

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

deletes all empty lines.

Probably the most useful command that can follow a global is the substitute command, for this can be used to make a change and print each affected line for verification. For example, we could change the word 'Unix' to 'UNIX' everywhere, and verify that it really worked, with

```
g/Unix/s//UNIX/gp
```

Notice that we used '/' in the substitute command to mean 'the previous pattern', in this case, 'Unix'. The **p** command is done on every line that matches the pattern, not just those on which a substitution took place.

The global command operates by making two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each marked line in turn is examined, dot is set to that line, and the command executed. This means that it is possible for the command that follows a **g** or **v** to use addresses, set dot, and so on, quite freely.

```
g/^\.PP/+
```


prints the line that follows each '.PP' command (the signal for a new paragraph in some formatting packages). Remember that '+' means 'one line past dot'. And

```
g/topic/?^\.SH?1
```

searches for each line that contains 'topic', scans backwards until it finds a line that begins '.SH' (a section heading) and prints the line that follows that, thus showing the section headings under which 'topic' is mentioned. Finally,

```
g/^\.EQ/+,/^\.EN/-p
```

prints all the lines that lie between lines beginning with '.EQ' and '.EN' formatting commands.

The **g** and **v** commands can also be preceded by line numbers, in which case the lines searched are only those in the range specified.

Multi-line Global Commands

It is possible to do more than one command under the control of a global command, although the syntax for expressing the operation is not especially natural or pleasant. As an example, suppose the task is to change 'x' to 'y' and 'a' to 'b' on all lines that contain 'thing'. Then

```
g/thing/s/x/y\  
s/a/b/
```

is sufficient. The '\' signals the **g** command that the set of commands continues on the next line; it terminates on the first line that does not end with '\'. (As a minor blemish, you can't use a substitute command to insert a newline within a **g** command.)

You should watch out for this problem: the command

```
g/x/s//y\  
s/a/b/
```

does *not* work as you expect. The remembered pattern is the last pattern that was actually executed, so sometimes it will be 'x' (as expected), and sometimes it will be 'a' (not expected). You must spell it out, like this:

```
g/x/s/x/y\  
s/a/b/
```

It is also possible to execute **a**, **c** and **i** commands under a global command; as with other multi-line constructions, all that is needed is to add a '\' at the end of each line except the last. Thus to add a '.nf' and '.sp' command before each '.EQ' line, type

```
g/^\.EQ/i\  
.nf\  
.sp
```

There is no need for a final line containing a '.' to terminate the **i** command, unless there are further commands being done under the global. On the other hand, it does no harm to put it in either.

5. CUT AND PASTE WITH UNIX COMMANDS

One editing area in which non-programmers seem not very confident is in what might be called 'cut and paste' operations — changing the name of a file, making a copy of a file somewhere else, moving a few lines from one place to another in a file, inserting one file in the middle of another, splitting a file into pieces, and splicing two or more files together.

Yet most of these operations are actually quite easy, if you keep your wits about you and go cautiously. The next several sections talk about cut and paste. We will begin with the UNIX commands for moving entire files around, then discuss **ed** commands for operating on pieces of files.

Changing the Name of a File

You have a file named 'memo' and you want it to be called 'paper' instead. How is it done?

The UNIX program that renames files is called **mv** (for 'move'); it 'moves' the file from one name to another, like this:

```
mv memo paper
```

That's all there is to it: **mv** from the old name to the new name.

```
mv oldname newname
```

Warning: if there is already a file around with the new name, its present contents will be silently clobbered by the information from the other file. The one exception is that you can't move a file to itself —

```
mv x x
```

is illegal.

Making a Copy of a File

Sometimes what you want is a copy of a file — an entirely fresh version. This might be because you want to work on a file, and yet save a copy in case something gets fouled up, or just because you're paranoid.

In any case, the way to do it is with the **cp** command. (**cp** stands for 'copy'; the system is big on short command names, which are appreciated by heavy users, but sometimes a strain for novices.) Suppose you have a file called 'good' and you want to save a copy before you make some dramatic editing changes. Choose a name — 'savegood' might be acceptable — then type

```
cp good savegood
```

This copies 'good' onto 'savegood', and you now have two identical copies of the file 'good'. (If 'savegood' previously contained something, it gets overwritten.)

Now if you decide at some time that you want to get back to the original state of 'good', you can say

```
mv savegood good
```

(if you're not interested in 'savegood' any more), or

```
cp savegood good
```

if you still want to retain a safe copy.

In summary, **mv** just renames a file; **cp** makes a duplicate copy. Both of them clobber the 'target' file if it already exists, so you had better be sure that's what you want to do *before* you do it.

Removing a File

If you decide you are really done with a file forever, you can remove it with the **rm** command:

```
rm savegood
```

throws away (irrevocably) the file called 'savegood'.

Putting Two or More Files Together

The next step is the familiar one of collecting two or more files into one big one. This will be needed, for example, when the author of a paper decides that several sections need to be combined into one. There are several ways to do it, of which the cleanest, once you get used to it, is a program called **cat**. (Not *all* programs have two-letter names.) **cat** is short for 'concatenate', which is exactly what we want to do.

Suppose the job is to combine the files 'file1' and 'file2' into a single file called 'bigfile'. If you say

```
cat file
```

the contents of 'file' will get printed on your terminal. If you say

```
cat file1 file2
```

the contents of 'file1' and then the contents of 'file2' will *both* be printed on your terminal, in that order. So **cat** combines the files, all right, but it's not much help to print them on the terminal — we want them in 'bigfile'.

Fortunately, there is a way. You can tell the system that instead of printing on your terminal, you want the same information put in a file. The way to do it is to add to the command line the character **>** and the name of the file where you want the output to go. Then you can say

```
cat file1 file2 >bigfile
```

and the job is done. (As with **cp** and **mv**, you're putting something into 'bigfile', and anything that was already there is destroyed.)

This ability to 'capture' the output of a program is one of the most useful aspects of the system. Fortunately it's not limited to the **cat** program — you can use it with *any* program that prints on your terminal. We'll see some more uses for it in a moment.

Naturally, you can combine several files, not just two:

```
cat file1 file2 file3 ... >bigfile
```

collects a whole bunch.

Question: is there any difference between

```
cp good savegood
```

and

```
cat good >savegood
```

Answer: for most purposes, no. You might reasonably ask why there are two programs in that case, since **cat** is obviously all you need. The answer is that **cp** will do some other things as well, which you can investigate for yourself by reading the manual. For now we'll stick to simple usages.

Adding Something to the End of a File

Sometimes you want to add one file to the end of another. We have enough building blocks now that you can do it; in fact before reading further it would be valuable if you figured out how. To be specific, how would you use **cp**, **mv** and/or **cat** to add the file 'good1' to the end of the file 'good'?

You could try

```
cat good good1 >temp
mv temp good
```

which is probably most direct. You should also understand why

```
cat good good1 >good
```

doesn't work. (Don't practice with a good 'good'!)

The easy way is to use a variant of **>**, called **>>**. In fact, **>>** is identical to **>** except that instead of clobbering the old file, it simply tacks stuff on at the end. Thus you could say

```
cat good1 >>good
```

and 'good1' is added to the end of 'good'. (And if 'good' didn't exist, this makes a copy of 'good1' called 'good'.)

6. CUT AND PASTE WITH THE EDITOR

Now we move on to manipulating pieces of files — individual lines or groups of lines. This is another area where new users seem unsure of themselves.

Filenames

The first step is to ensure that you know the **ed** commands for reading and writing files. Of course you can't go very far without knowing **r** and **w**. Equally useful, but less well known, is the 'edit' command **e**. Within **ed**, the command

```
e newfile
```

says 'I want to edit a new file called *newfile*, without leaving the editor.' The **e** command discards whatever you're currently working on and starts over on *newfile*. It's exactly the same as if you had quit with the **q** command, then re-entered **ed** with a new file name, except that if you have a pattern remembered, then a command like **//** will still work.

If you enter **ed** with the command

```
ed file
```

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

```

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

```

(and so on) does a series of edits on various files without ever leaving **ed** and without typing the name of any file more than once. (As an aside, if you examine the sequence of commands here, you can see why many UNIX systems use **e** as a synonym for **ed**.)

You can find out the remembered file name at any time with the **f** command; just type **f** without a file name. You can also change the name of the remembered file name with **f**; a useful sequence is

```

ed precious
f junk
... (editing) ...

```

which gets a copy of a precious file, then uses **f** to guarantee that a careless **w** command won't clobber the original.

Inserting One File into Another

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

```
Table 1 shows that ...
```

and the data contained in 'table' has to go there, probably so it will be formatted properly by **nroff** or **troff**. Now what?

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

```

ed memo
/Table 1/
Table 1 shows that ... [response from ed]
.r table

```

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

Writing out Part of a File

The other side of the coin is writing out part of the document you're editing. For example, maybe you want to split out into a separate file that table from the previous example, so it can be formatted and tested separately. Suppose that in the file being edited we have

```

.TS
...[lots of stuff]
.TE

```

which is the way a table is set up for the **tbl** program. To isolate the table in a

separate file called 'table', first find the start of the table (the '.TS' line), then write out the interesting part:

```

/^\.TS/
.TS [ed prints the line it found]
./^\.TE/w table

```

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

```

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

```

The point is that the **w** command can write out a group of lines, instead of the whole file. In fact, you can write out a single line if you like; just give one line number instead of two. For example, if you have just typed a horribly complicated line and you know that it (or something like it) is going to be needed later, then save it — don't re-type it. In the editor, say

```

a
...lots of stuff...
...horrible line...
.
.w temp
a
...more stuff...
.
.r temp
a
...more stuff...
.

```

This last example is worth studying, to be sure you appreciate what's going on.

Moving Lines Around

Suppose you want to move a paragraph from its present position in a paper to the end. How would you do it? As a concrete example, suppose each paragraph in the paper begins with the formatting command '.PP'. Think about it and write down the details before reading on.

The brute force way (not necessarily bad) is to write the paragraph onto a temporary file, delete it from its current position, then read in the temporary file at the end. Assuming that you are sitting on the '.PP' command that begins the paragraph, this is the sequence of commands:

```

./^\.PP/-w temp
./-d
$r temp

```

That is, from where you are now ('.') until one line before the next '.PP' ('/^\.PP/-') write onto 'temp'. Then delete the same lines. Finally, read 'temp' at the end.

As we said, that's the brute force way. The easier way (often) is to use the *move* command **m** that **ed** provides — it lets you do the whole set of operations at one crack, without any temporary file.

The **m** command is like many other **ed** commands in that it takes up to two line numbers in front that tell what lines are to be affected. It is also *followed* by a line number that tells where the lines are to go. Thus

```
line1, line2 m line3
```

says to move all the lines between 'line1' and 'line2' after 'line3'. Naturally, any of 'line1' etc., can be patterns between slashes, \$ signs, or other ways to specify lines.

Suppose again that you're sitting at the first line of the paragraph. Then you can say

```
./^\.PP/-m$
```

That's all.

As another example of a frequent operation, you can reverse the order of two adjacent lines by moving the first one to after the second. Suppose that you are positioned at the first. Then

```
m+
```

does it. It says to move line dot to after one line after line dot. If you are positioned on the second line,

```
m-
```

does the interchange.

As you can see, the **m** command is more succinct and direct than writing, deleting and re-reading. When is brute force better anyway? This is a matter of personal taste — do what you have most confidence in. The main difficulty with the **m** command is that if you use patterns to specify both the lines you are moving and the target, you have to take care that you specify them properly, or you may well not move the lines you thought you did. The result of a botched **m** command can be a ghastly mess. Doing the job a step at a time makes it easier for you to verify at each step that you accomplished what you wanted to. It's also a good idea to issue a **w** command before doing anything complicated; then if you goof, it's easy to back up to where you were.

Marks

ed provides a facility for marking a line with a particular name so you can later reference it by name regardless of its actual line number. This can be handy for moving lines, and for keeping track of them as they move. The *mark* command is **k**; the command

```
kx
```

marks the current line with the name 'x'. If a line number precedes the **k**, that line is marked. (The mark name must be a single lower case letter.) Now you can refer to the marked line with the address

```
' x
```

Marks are most useful for moving things around. Find the first line of the block to be moved, and mark it with ' a. Then find the last line and mark it with ' b. Now position yourself at the place where the stuff is to go and say

' a,' bm.

Bear in mind that only one line can have a particular mark name associated with it at any given time.

Copying Lines

We mentioned earlier the idea of saving a line that was hard to type or used often, so as to cut down on typing time. Of course this could be more than one line; then the saving is presumably even greater.

ed provides another command, called **t** (for 'transfer') for making a copy of a group of one or more lines at any point. This is often easier than writing and reading.

The **t** command is identical to the **m** command, except that instead of moving lines it simply duplicates them at the place you named. Thus

```
1,$t$
```

duplicates the entire contents that you are editing. A more common use for **t** is for creating a series of lines that differ only slightly. For example, you can say

```
a
..... x ..... (long line)
.
t.          (make a copy)
s/x/y/     (change it a bit)
t.          (make third copy)
s/y/z/     (change it a bit)
```

and so on.

The Temporary Escape '!'

Sometimes it is convenient to be able to temporarily escape from the editor to do some other UNIX command, perhaps one of the file copy or move commands discussed in section 5, without leaving the editor. The 'escape' command **!** provides a way to do this.

If you say

```
!any UNIX command
```

your current editing state is suspended, and the UNIX command you asked for is executed. When the command finishes, **ed** will signal you by printing another **!**; at that point you can resume editing.

You can really do *any* UNIX command, including another **ed**. (This is quite common, in fact.) In this case, you can even do another **!**.

7. SUPPORTING TOOLS

There are several tools and techniques that go along with the editor, all of which are relatively easy once you know how **ed** works, because they are all based on the editor. In this section we will give some fairly cursory examples of these tools, more to indicate their existence than to provide a complete tutorial. More information on each can be found in [3].

Grep

Sometimes you want to find all occurrences of some word or pattern in a set of files, to edit them or perhaps just to verify their presence or absence. It may be possible to edit each file separately and look for the pattern of interest, but if there are many files this can get very tedious, and if the files are really big, it may be impossible because of limits in **ed**.

The program **grep** was invented to get around these limitations. The search patterns that we have described in the paper are often called 'regular expressions', and 'grep' stands for

```
g/re/p
```

That describes exactly what **grep** does — it prints every line in a set of files that contains a particular pattern. Thus

```
grep 'thing' file1 file2 file3 ...
```

finds 'thing' wherever it occurs in any of the files 'file1', 'file2', etc. **grep** also indicates the file in which the line was found, so you can later edit it if you like.

The pattern represented by 'thing' can be any pattern you can use in the editor, since **grep** and **ed** use exactly the same mechanism for pattern searching. It is wisest always to enclose the pattern in the single quotes '...!' if it contains any non-alphabetic characters, since many such characters also mean something special to the UNIX command interpreter (the 'shell'). If you don't quote them, the command interpreter will try to interpret them before **grep** gets a chance.

There is also a way to find lines that *don't* contain a pattern:

```
grep -v 'thing' file1 file2 ...
```

finds all lines that don't contain 'thing'. The **-v** must occur in the position shown. Given **grep** and **grep -v**, it is possible to do things like selecting all lines that contain some combination of patterns. For example, to get all lines that contain 'x' but not 'y':

```
grep x file... | grep -v y
```

(The notation | is a 'pipe', which causes the output of the first command to be used as input to the second command; see [2].)

Editing Scripts

If a fairly complicated set of editing operations is to be done on a whole set of files, the easiest thing to do is to make up a 'script', i.e., a file that contains the operations you want to perform, then apply this script to each file in turn.

For example, suppose you want to change every 'Unix' to 'UNIX' and every 'Gcos' to 'GCOS' in a large number of files. Then put into the file 'script' the lines

```
g/Unix/s//UNIX/g
g/Gcos/s//GCOS/g
w
q
```

Now you can say

```
ed file1 <script
ed file2 <script
...
```

This causes **ed** to take its commands from the prepared script. Notice that the whole job has to be planned in advance.

And of course by using the UNIX command interpreter, you can cycle through a set of files automatically, with varying degrees of ease.

Sed

sed ('stream editor') is a version of the editor with restricted capabilities but which is capable of processing unlimited amounts of input. Basically **sed** copies its input to its output, applying one or more editing commands to each line of input.

As an example, suppose that we want to do the 'Unix' to 'UNIX' part of the example given above, but without rewriting the files. Then the command

```
sed 's/Unix/UNIX/g' file1 file2 ...
```

applies the command 's/Unix/UNIX/g' to all lines from 'file1', 'file2', etc., and copies all lines to the output. The advantage of using **sed** in such a case is that it can be used with input too large for **ed** to handle. All the output can be collected in one place, either in a file or perhaps piped into another program.

If the editing transformation is so complicated that more than one editing command is needed, commands can be supplied from a file, or on the command line, with a slightly more complex syntax. To take commands from a file, for example,

```
sed -f cmdfile input-files...
```

sed has further capabilities, including conditional testing and branching, which we cannot go into here.

References

- [1] Brian W. Kernighan, *A Tutorial Introduction to the UNIX Text Editor*, Bell Laboratories internal memorandum.
- [2] Brian W. Kernighan, *UNIX For Beginners*, Bell Laboratories internal memorandum.
- [3] Ken L. Thompson and Dennis M. Ritchie, *The UNIX Programmer's Manual*. Bell Laboratories.

Appendix B: Writing Papers With nroff Using -me

Eric P. Allman

Electronics Research Laboratory
University of California, Berkeley
Berkeley, California 94720

This document describes the text processing facilities available on the UNIX operating system via NROFF and the -me macro package. It is assumed that the reader already is generally familiar with the UNIX operating system and a text editor such as *ex*. This is intended to be a casual introduction, and as such not all material is covered. In particular, many variations and additional features of the -me macro package are not explained. For a complete discussion of this and other issues, see *The -me Reference Manual* and *The NROFF/TROFF Reference Manual*.

NROFF, a computer program that runs on the UNIX operating system, reads an input file prepared by the user and outputs a formatted paper suitable for publication or framing. The input consists of *text*, or words to be printed, and *requests*, which give instructions to the NROFF program telling how to format the printed copy.

Section 1 describes the basics of text processing. Section 2 describes the basic requests. Section 3 introduces displays. Annotations, such as footnotes, are handled in section 4. The more complex requests which are not discussed in section 2 are covered in section 5. Finally, section 6 discusses things you will need to know if you want to typeset documents. If you are a novice, you probably won't want to read beyond section 4 until you have tried some of the basic features out.

When you have your raw text ready, call the NROFF formatter by typing as a request to the UNIX shell:

```
nroff -me -Ttype files
```

where *type* describes the type of terminal you are outputting to. Common values are *dtc* for a DTC 300s (daisy-wheel type) printer and *lpr* for the line printer. If the -T flag is omitted, a "lowest common denominator" terminal is assumed; this is good for previewing output on most terminals. A complete description of options to the NROFF command can be found in *The NROFF/TROFF Reference Manual*.

The word *argument* is used in this manual to mean a word or number which appears on the same line as a request which modifies the meaning of that request. For example, the request

```
.sp
```

spaces one line, but

```
.sp 4
```

spaces four lines. The number 4 is an *argument* to the `.sp` request which says to space four lines instead of one. Arguments are separated from the request and from each other by spaces.

1. Basics of Text Processing

The primary function of NROFF is to *collect* words from input lines, *fill* output lines with those words, *justify* the right hand margin by inserting extra spaces in the line, and output the result. For example, the input:

```
Now is the time  
for all good men  
to come to the aid  
of their party.  
Four score and seven  
years ago,...
```

will be read, packed onto output lines, and justified to produce:

```
Now is the time for all good men to come to the aid of their party.  
Four score and seven years ago,...
```

Sometimes you may want to start a new output line even though the line you are on is not yet full; for example, at the end of a paragraph. To do this you can cause a *break*, which starts a new output line. Some requests cause a break automatically, as do blank input lines and input lines beginning with a space.

Not all input lines are text to be formatted. Some of the input lines are *requests* which describe how to format the text. Requests always have a period or an apostrophe (“ ’ ”) as the first character of the input line.

The text formatter also does more complex things, such as automatically numbering pages, skipping over page folds, putting footnotes in the correct place, and so forth.

I can offer you a few hints for preparing text for input to NROFF. First, keep the input lines short. Short input lines are easier to edit, and NROFF will pack words onto longer lines for you anyhow. In keeping with this, it is helpful to begin a new line after every period, comma, or phrase, since common corrections are to add or delete sentences or phrases. Second, do not put spaces at the end of lines, since this can sometimes confuse the NROFF processor. Third, do not hyphenate words at the end of lines (except words that should have hyphens in them, such as “mother-in-law”); NROFF is smart

enough to hyphenate words for you as needed, but is not smart enough to take hyphens out and join a word back together. Also, words such as "mother-in-law" should not be broken over a line, since then you will get a space where not wanted, such as "mother- in-law".

2. Basic Requests

2.1. Paragraphs

Paragraphs are begun by using the `.pp` request. For example, the input:

```
.pp
Now is the time for all good men
to come to the aid of their party.
Four score and seven years ago,...
```

produces a blank line followed by an indented first line. The result is:

```
    Now is the time for all good men to come to the aid of their
    party. Four score and seven years ago,...
```

Notice that the sentences of the paragraphs *must not* begin with a space, since blank lines and lines beginning with spaces cause a break. For example, if I had typed:

```
.pp
Now is the time for all good men
    to come to the aid of their party.
Four score and seven years ago,...
```

The output would be:

```
    Now is the time for all good men
    to come to the aid of their party. Four score and seven years
ago,...
```

A new line begins after the word "men" because the second line began with a space character.

There are many fancier types of paragraphs, which will be described later.

2.2. Headers and Footers

Arbitrary headers and footers can be put at the top and bottom of every page. Two requests of the form `.he title` and `.fo title` define the titles to put at the head and the foot of every page, respectively. The titles are called *three-part* titles, that is, there is a left-justified part, a centered part, and a right-justified part. To separate these three parts the first character of *title* (whatever it may be) is used as a delimiter. Any character may be used, but backslash and double quote marks should be avoided. The percent sign is replaced by the current page number whenever found in the

title. For example, the input:

```
.he ``%``
.fo `Jane Jones` `My Book`
```

results in the page number centered at the top of each page, "Jane Jones" in the lower left corner, and "My Book" in the lower right corner.

2.3. Double Spacing

NROFF will double space output text automatically if you use the request `.ls 2`, as is done in this section. You can revert to single spaced mode by typing `.ls 1`.

2.4. Page Layout

A number of requests allow you to change the way the printed copy looks, sometimes called the *layout* of the output page. Most of these requests adjust the placing of "white space" (blank lines or spaces). In these explanations, characters in italics should be replaced with values you wish to use; bold characters represent characters which should actually be typed.

The `.bp` request starts a new page.

The request `.sp N` leaves N lines of blank space. N can be omitted (meaning skip a single line) or can be of the form Ni (for N inches) or Nc (for N centimeters). For example, the input:

```
.sp 1.5i
My thoughts on the subject
.sp
```

leaves one and a half inches of space, followed by the line "My thoughts on the subject", followed by a single blank line.

The `.in +N` request changes the amount of white space on the left of the page (the *indent*). The argument N can be of the form $+N$ (meaning leave N spaces more than you are already leaving), $-N$ (meaning leave less than you do now), or just N (meaning leave exactly N spaces). N can be of the form Ni or Nc also. For example, the input:

```
initial text
.in 5
some text
.in +1i
more text
.in -2c
final text
```

produces "some text" indented exactly five spaces from the left margin, "more text" indented five spaces plus one inch from the left margin (fifteen

spaces on a pica typewriter), and "final text" indented five spaces plus one inch minus two centimeters from the margin. That is, the output is:

```

initial text
      some text
            more text
                  final text

```

The `.ti +N` (temporary indent) request is used like `.in +N` when the indent should apply to one line only, after which it should revert to the previous indent. For example, the input:

```

.in li
.ti 0
Ware, James R. The Best of Confucius,
Halcyon House, 1950.
An excellent book containing translations of
most of Confucius' most delightful sayings.
A definite must for anyone interested in the early foundations
of Chinese philosophy.

```

produces:

```

Ware, James R. The Best of Confucius, Halcyon House, 1950. An excellent
      book containing translations of most of Confucius' most
      delightful sayings. A definite must for anyone interested in
      the early foundations of Chinese philosophy.

```

Text lines can be centered by using the `.ce` request. The line after the `.ce` is centered (horizontally) on the page. To center more than one line, use `.ce N` (where *N* is the number of lines to center), followed by the *N* lines. If you want to center many lines but don't want to count them, type:

```

.ce 1000
lines to center
.ce 0

```

The `.ce 0` request tells NROFF to center zero more lines, in other words, stop centering.

All of these requests cause a break; that is, they always start a new line. If you want to start a new line without performing any other action, use `.br`.

2.5. Underlining

Text can be underlined using the `.ul` request. The `.ul` request causes the next input line to be underlined when output. You can underline multiple lines by stating a count of *input* lines to underline, followed by those lines (as with the `.ce` request). For example, the input:

.ul 2

Notice that these two input lines
are underlined.

will underline those eight words in NROFF. (In TROFF they will be set in italics.)

3. Displays

Displays are sections of text to be set off from the body of the paper. Major quotes, tables, and figures are types of displays, as are all the examples used in this document. All displays except centered blocks are output single spaced.

3.1. Major Quotes

Major quotes are quotes which are several lines long, and hence are set in from the rest of the text without quote marks around them. These can be generated using the commmands `.(q` and `.)q` to surround the quote. For example, the input:

As Weizenbaum points out:

```
.(q
It is said that to explain is to explain away.
This maxim is nowhere so well fulfilled
as in the areas of computer programming,...
.)q
```

generates as output:

As Weizenbaum points out:

```
It is said that to explain is to explain away. This maxim is nowhere so
well fulfilled as in the areas of computer programming,...
```

3.2. Lists

A *list* is an indented, single spaced, unfilled display. Lists should be used when the material to be printed should not be filled and justified like normal text, such as columns of figures or the examples used in this paper. Lists are surrounded by the requests `.(l` and `.)l`. For example, type:

Alternatives to avoid deadlock are:

```
.(l
Lock in a specified order
Detect deadlock and back out one process
Lock all resources needed before proceeding
.)l
```

will produce:

Alternatives to avoid deadlock are:

```
Lock in a specified order
Detect deadlock and back out one process
```


Lock all resources needed before proceeding

3.3. Keeps

A *keep* is a display of lines which are kept on a single page if possible. An example of where you would use a keep might be a diagram. Keeps differ from lists in that lists may be broken over a page boundary whereas keeps will not.

Blocks are the basic kind of keep. They begin with the request `.(b` and end with the request `.)b`. If there is not room on the current page for everything in the block, a new page is begun. This has the unpleasant effect of leaving blank space at the bottom of the page. When this is not appropriate, you can use the alternative, called *floating keeps*.

Floating keeps move relative to the text. Hence, they are good for things which will be referred to by name, such as "See figure 3". A floating keep will appear at the bottom of the current page if it will fit; otherwise, it will appear at the top of the next page. Floating keeps begin with the line `.(z` and end with the line `.)z`. For an example of a floating keep, see figure 1. The `.hl` request is used to draw a horizontal line so that the figure stands out from the text.

3.4. Fancier Displays

Keeps and lists are normally collected in *nofill* mode, so that they are good for tables and such. If you want a display in fill mode (for text), type `.(l F` (Throughout this section, comments applied to `.(l` also apply to `.(b` and `.(z`). This kind of display will be indented from both margins. For example, the input:

```
.(z
.hl
Text of keep to be floated.
.sp
.ce
Figure 1. Example of a Floating Keep.
.hl
.)z
```

Figure 1. Example of a Floating Keep.

```
.(l F
And now boys and girls,
a newer, bigger, better toy than ever before!
Be the first on your block to have your own computer!
Yes kids, you too can have one of these modern
data processing devices.
You too can produce beautifully formatted papers
without even batting an eye!
.)l
```

will be output as:

```
And now boys and girls, a newer, bigger, better toy than ever be-
fore! Be the first on your block to have your own computer! Yes
kids, you too can have one of these modern data processing devices.
You too can produce beautifully formatted papers without even bat-
ting an eye!
```

Lists and blocks are also normally indented (floating keeps are normally left justified). To get a left-justified list, type `.(l L`. To get a list centered line-for-line, type `.(l C`. For example, to get a filled, left justified list, enter:

```
.(l L F
text of block
.)l
```

The input:

```
.(l
first line of unfilled display
more lines
.)l
```

produces the indented text:

```
first line of unfilled display
more lines
```

Typing the character `L` after the `.(l` request produces the left justified result:

```
first line of unfilled display
more lines
```

Using `C` instead of `L` produces the line-at-a-time centered output:

```
first line of unfilled display
more lines
```

Sometimes it may be that you want to center several lines as a group, rather than centering them one line at a time. To do this use centered blocks, which are surrounded by the requests `.(c` and `.)c`. All the lines are centered as a unit, such that the longest line is centered and the rest are

lined up around that line. Notice that lines do not move relative to each other using centered blocks, whereas they do using the **C** argument to keeps.

Centered blocks are *not* keeps, and may be used in conjunction with keeps. For example, to center a group of lines as a unit and keep them on one page, use:

```
.(b L
.(c
first line of unfilled display
more lines
.)c
.)b
```

to produce:

```
first line of unfilled display
more lines
```

If the block requests **.(b** and **.)b** had been omitted the result would have been the same, but with no guarantee that the lines of the centered block would have all been on one page. Note the use of the **L** argument to **.(b**; this causes the centered block to center within the entire line rather than within the line minus the indent. Also, the center requests must be nested *inside* the keep requests.

4. Annotations

There are a number of requests to save text for later printing. *Footnotes* are printed at the bottom of the current page. *Delayed text* is intended to be a variant form of footnote; the text is printed only when explicitly called for, such as at the end of each chapter. *Indexes* are a type of delayed text having a tag (usually the page number) attached to each entry after a row of dots. Indexes are also saved until called for explicitly.

4.1. Footnotes

Footnotes begin with the request **.(f** and end with the request **.)f**. The current footnote number is maintained automatically, and can be used by typing ******, to produce a footnote number¹. The number is automatically incremented after every footnote. For example, the input:

¹Like this.

```

.(q
A man who is not upright
and at the same time is presumptuous;
one who is not diligent and at the same time is ignorant;
one who is untruthful and at the same time is incompetent;
such men I do not count among acquaintances.\**
.(f
\**James R. Ware,
.ul
The Best of Confucius,
Halcyon House, 1950.
Page 77.
.)f
.)q

```

generates the result:

A man who is not upright and at the same time is presumptuous; one who is not diligent and at the same time is ignorant; one who is untruthful and at the same time is incompetent; such men I do not count among acquaintances.²

It is important that the footnote appears *inside* the quote, so that you can be sure that the footnote will appear on the same page as the quote.

4.2. Delayed Text

Delayed text is very similar to a footnote except that it is printed when called for explicitly. This allows a list of references to appear (for example) at the end of each chapter, as is the convention in some disciplines. Use `*#` on delayed text instead of `**` as on footnotes.

If you are using delayed text as your standard reference mechanism, you can still use footnotes, except that you may want to reference them with special characters* rather than numbers.

4.3. Indexes

An "index" (actually more like a table of contents, since the entries are not sorted alphabetically) resembles delayed text, in that it is saved until called for. However, each entry has the page number (or some other tag) appended to the last line of the index entry after a row of dots.

Index entries begin with the request `.(x` and end with `.)x`. The `.)x` request may have a argument, which is the value to print as the "page number". It defaults to the current page number. If the page number given is an underscore ("_") no page number or line of dots is printed at all. To get the line of dots without a page number, type `.)x ""`, which

²James R. Ware, *The Best of Confucius*, Halcyon House, 1950. Page 77.

*Such as an asterisk.

specifies an explicitly null page number.

The `.xp` request prints the index.

For example, the input:

```
.(x
Sealing wax
.)x
.(x
Cabbages and kings
.)x _
.(x
Why the sea is boiling hot
.)x 2.5a
.(x
Whether pigs have wings
.)x ""
.(x
This is a terribly long index entry, such as might be used
for a list of illustrations, tables, or figures; I expect it to
take at least two lines.
.)x
.xp
```

generates:

```
Sealing wax ..... 11
Cabbages and kings
Why the sea is boiling hot ..... 2.5a
Whether pigs have wings .....
This is a terribly long index entry, such as might be used
of illustrations, tables, or figures; I expect it to take at least
two lines. .... 11
```

The `.(x` request may have a single character argument, specifying the "name" of the index; the normal index is `x`. Thus, several "indicies" may be maintained simultaneously (such as a list of tables, table of contents, etc.).

Notice that the index must be printed at the *end* of the paper, rather than at the beginning where it will probably appear (as a table of contents); the pages may have to be physically rearranged after printing.

5. Fancier Features

A large number of fancier requests exist, notably requests to provide other sorts of paragraphs, numbered sections of the form **1.2.3** (such as used in this document), and multicolumn output.

5.1. More Paragraphs

Paragraphs generally start with a blank line and with the first line indented. It is possible to get left-justified block-style paragraphs by using `.lp` instead of `.pp`, as demonstrated by the next paragraph.

Sometimes you want to use paragraphs that have the *body* indented, and the first line exdented (opposite of indented) with a label. This can be done with the `.ip` request. A word specified on the same line as `.ip` is printed in the margin, and the body is lined up at a prespecified position (normally five spaces). For example, the input:

```
.ip one
This is the first paragraph.
Notice how the first line
of the resulting paragraph lines up
with the other lines in the paragraph.
.ip two
And here we are at the second paragraph already.
You may notice that the argument to .ip
appears
in the margin.
.lp
We can continue text...
```

produces as output:

one This is the first paragraph. Notice how the first line of the resulting paragraph lines up with the other lines in the paragraph.

two And here we are at the second paragraph already. You may notice that the argument to `.ip` appears in the margin.

We can continue text without starting a new indented paragraph by using the `.lp` request.

If you have spaces in the label of a `.ip` request, you must use an “unpaddable space” instead of a regular space. This is typed as a backslash character (“\”) followed by a space. For example, to print the label “Part 1”, enter:

```
.ip "Part\ 1"
```

If a label of an indented paragraph (that is, the argument to `.ip`) is longer than the space allocated for the label, `.ip` will begin a new line after the label. For example, the input:

```
.ip longlabel
This paragraph had a long label.
The first character of text on the first line
will not line up with the text on second and subsequent lines,
although they will line up with each other.
```

will produce:

longlabel

This paragraph had a long label. The first character of text on the first line will not line up with the text on second and subsequent lines, although they will line up with each other.

It is possible to change the size of the label by using a second argument which is the size of the label. For example, the above example could be done correctly by saying:

```
.ip longlabel 10
```

which will make the paragraph indent 10 spaces for this paragraph only. If you have many paragraphs to indent all the same amount, use the *number register ii*. For example, to leave one inch of space for the label, type:

```
.nr ii li
```

somewhere before the first call to **.ip**. Refer to the troff reference manual (in Section 2) for more information.

If **.ip** is used with no argument at all no hanging tag will be printed. For example, the input:

```
.ip [a]
```

This is the first paragraph of the example.

We have seen this sort of example before.

```
.ip
```

This paragraph is lined up with the previous paragraph, but it has no tag in the margin.

produces as output:

```
[a] This is the first paragraph of the example. We have seen this sort of
example before.
```

This paragraph is lined up with the previous paragraph, but it has no tag in the margin.

A special case of **.ip** is **.np**, which automatically numbers paragraphs sequentially from 1. The numbering is reset at the next **.pp**, **.lp**, or **.sh** (to be described in the next section) request. For example, the input:

```
.np
This is the first point.
.np
This is the second point.
Points are just regular paragraphs
which are given sequence numbers automatically
by the .np request.
.pp
This paragraph will reset numbering by .np.
.np
For example,
we have reverted to numbering from one now.
```

generates:

- (1) This is the first point.
- (2) This is the second point. Points are just regular paragraphs which are given sequence numbers automatically by the .np request.
This paragraph will reset numbering by .np.
- (1) For example, we have reverted to numbering from one now.

5.2. Section Headings

Section numbers (such as the ones used in this document) can be automatically generated using the `.sh` request. You must tell `.sh` the *depth* of the section number and a section title. The depth specifies how many numbers are to appear (separated by decimal points) in the section number. For example, the section number **4.2.5** has a depth of three.

Section numbers are incremented in a fairly intuitive fashion. If you add a number (increase the depth), the new number starts out at one. If you subtract section numbers (or keep the same number) the final number is incremented. For example, the input:

```
.sh 1 "The Preprocessor"
.sh 2 "Basic Concepts"
.sh 2 "Control Inputs"
.sh 3
.sh 3
.sh 1 "Code Generation"
.sh 3
```

produces as output the result:

1. The Preprocessor

1.1. Basic Concepts

1.2. Control Inputs

1.2.1.

1.2.2.

2. Code Generation

2.1.1.

You can specify the section number to begin by placing the section number after the section title, using spaces instead of dots. For example, the request:

```
.sh 3 "Another section" 7 3 4
```

will begin the section numbered **7.3.4**; all subsequent `.sh` requests will number relative to this number.

There are more complex features which will cause each section to be indented proportionally to the depth of the section. For example, if you enter:

```
.nr si N
```

each section will be indented by an amount N . N must have a scaling factor attached, that is, it must be of the form Nx , where x is a character telling what units N is in. Common values for x are `i` for inches, `c` for centimeters, and `n` for *ens* (the width of a single character). For example, to indent each section one-half inch, type:

```
.nr si 0.5i
```

After this, sections will be indented by one-half inch per level of depth in the section number. For example, this document was produced using the request

```
.nr si 3n
```

at the beginning of the input file, giving three spaces of indent per section depth.

Section headers without automatically generated numbers can be done using:

```
.uh "Title"
```

which will do a section heading, but will put no number on the section.

5.3. Parts of the Basic Paper

There are some requests which assist in setting up papers. The `.tp` request initializes for a title page. There are no headers or footers on a title page, and unlike other pages you can space down and leave blank space at the top. For example, a typical title page might appear as:

```
.tp
.sp 2i
.(l C
THE GROWTH OF TOENAILS
IN UPPER PRIMATES
.sp
by
.sp
Frank N. Furter
.)l
.bp
```

The request `.th` sets up the environment of the NROFF processor to do a thesis, using the rules established at Berkeley. It defines the correct headers and footers (a page number in the upper right hand corner only), sets the margins correctly, and double spaces.

The `."+c T` request can be used to start chapters. Each chapter is automatically numbered from one, and a heading is printed at the top of each chapter with the chapter number and the chapter name *T*. For example, to begin a chapter called "Conclusions", use the request:

```
."+c "CONCLUSIONS"
```

which will produce, on a new page, the lines

```
CHAPTER 5
CONCLUSIONS
```

with appropriate spacing for a thesis. Also, the header is moved to the foot of the page on the first page of a chapter. Although the `."+c` request was not designed to work only with the `.th` request, it is tuned for the format acceptable for a PhD thesis at Berkeley.

If the title parameter *T* is omitted from the `."+c` request, the result is a chapter with no heading. This can also be used at the beginning of a paper; for example, `."+c` was used to generate page one of this document.

Although papers traditionally have the abstract, table of contents, and so forth at the front of the paper, it is more convenient to format and print them last when using NROFF. This is so that index entries can be collected and then printed for the table of contents (or whatever). At the end of the paper, issue the `."+ P` request, which begins the preliminary part of the paper. After issuing this request, the `."+c` request will begin a preliminary section of the paper. Most notably, this prints the page number restarted from one in lower case Roman numbers. `."+c` may be used repeatedly to begin different parts of the front material for example, the abstract, the table of contents, acknowledgments, list of illustrations, etc. The request `."+ B` may also be used to begin the bibliographic section at the end of the paper. For example, the paper might appear as outlined in figure 2. (In this figure, comments begin with the sequence `\`.)

```

.th                \| set for thesis mode
.fo ' 'DRAFT' '   \| define footer for each page
.tp                \| begin title page
.(l C              \| center a large block
THE GROWTH OF TOENAILS
IN UPPER PRIMATES
.sp
by
.sp
Frank Furter
.)l                \| end centered part
.+c INTRODUCTION  \| begin chapter named "INTRODUCTION"
.(x t              \| make an entry into index 't'
Introduction
.)x                \| end of index entry
text of chapter one
.+c "NEXT CHAPTER" \| begin another chapter
.(x t              \| enter into index 't' again
Next Chapter
.)x
text of chapter two
.+c CONCLUSIONS
.(x t
Conclusions
.)x
text of chapter three
.++ B              \| begin bibliographic information
.+c BIBLIOGRAPHY  \| begin another 'chapter'
.(x t
Bibliography
.)x
text of bibliography
.++ P              \| begin preliminary material
.+c "TABLE OF CONTENTS"
.xp t              \| print index 't' collected above
.+c PREFACE        \| begin another preliminary section
text of preface

```

Figure 2. Outline of a Sample Paper

5.4. Equations and Tables

Two special UNIX programs exist to format special types of material. **Eqn** and **neqn** set equations for the phototypesetter and NROFF respectively. **Tbl** arranges to print extremely pretty tables in a variety of formats. This document will only describe the embellishments to the standard features; consult the reference manuals for those processors for a description of their use.

The **eqn** and **neqn** programs are described fully in the document *Typesetting Mathematics - Users' Guide* by Brian W. Kernighan and Lorinda L. Cherry. Equations are centered, and are kept on one page. They are introduced by the **.EQ** request and terminated by the **.EN** request.

The **.EQ** request may take an equation number as an optional argument, which is printed vertically centered on the right hand side of the equation. If the equation becomes too long it should be split between two lines. To do this, type:

```
.EQ (eq 34)
text of equation 34
.EN C
.EQ
continuation of equation 34
.EN
```

The **C** on the **.EN** request specifies that the equation will be continued.

The **tbl** program produces tables. It is fully described (including numerous examples) in the document *Tbl - A Program to Format Tables* by M. E. Lesk. Tables begin with the **.TS** request and end with the **.TE** request. Tables are normally kept on a single page. If you have a table which is too big to fit on a single page, so that you know it will extend to several pages, begin the table with the request **.TS H** and put the request **.TH** after the part of the table which you want duplicated at the top of every page that the table is printed on. For example, a table definition for a long table might look like:

```
.TS H
c s s
n n n.
THE TABLE TITLE
.TH
text of the table
.TE
```

5.5. Two Column Output

You can get two column output automatically by using the request **.2c**. This causes everything after it to be output in two-column form. The request **.bc** will start a new column; it differs from **.bp** in that **.bp** may

leave a totally blank column when it starts a new page. To revert to single column output, use **.1c**.

5.6. Defining Macros

A *macro* is a collection of requests and text which may be used by stating a simple request. Macros begin with the line **.de xx** (where *xx* is the name of the macro to be defined) and end with the line consisting of two dots. After defining the macro, stating the line **.xx** is the same as stating all the other lines. For example, to define a macro that spaces 3 lines and then centers the next input line, enter:

```
.de SS
.sp 3
.ce
..
```

and use it by typing:

```
.SS
Title Line
(beginning of text)
```

Macro names may be one or two characters. In order to avoid conflicts with names in **-me**, always use upper case letters as names. The only names to avoid are **TS**, **TH**, **TE**, **EQ**, and **EN**.

5.7. Annotations Inside Keeps

Sometimes you may want to put a footnote or index entry inside a keep. For example, if you want to maintain a "list of figures" you will want to do something like:

```
.(z
.(c
text of figure
.)c
.ce
Figure 5.
.(x f
Figure 5
.)x
.)z
```

which you may hope will give you a figure with a label and an entry in the index **f** (presumably a list of figures index). Unfortunately, the index entry is read and interpreted when the keep is read, not when it is printed, so the page number in the index is likely to be wrong. The solution is to use the magic string **\!** at the beginning of all the lines dealing with the index. In other words, you should use:

```

.(z
.(c
Text of figure
.)c
.ce
Figure 5.
\!(x f
\!Figure 5
\!.)x
.)z

```

which will defer the processing of the index until the figure is output. This will guarantee that the page number in the index is correct. The same comments apply to blocks (with **.(b** and **.)b**) as well.

6. TROFF and the Photosetter

With a little care, you can prepare documents that will print nicely on either a regular terminal or when phototypeset using the TROFF formatting program.

6.1. Fonts

A *font* is a style of type. There are three fonts that are available simultaneously, Times Roman, Times Italic, and Times Bold, plus the special math font. The normal font is Roman. Text which would be underlined in NROFF with the **.ul** request is set in italics in TROFF.

There are ways of switching between fonts. The requests **.r**, **.i**, and **.b** switch to Roman, italic, and bold fonts respectively. You can set a single word in some font by typing (for example):

```
.i word
```

which will set *word* in italics but does not affect the surrounding text. In NROFF, italic and bold text is underlined.

Notice that if you are setting more than one word in whatever font, you must surround that word with double quote marks (‘ ’’) so that it will appear to the NROFF processor as a single word. The quote marks will not appear in the formatted text. If you do want a quote mark to appear, you should quote the entire string (even if a single word), and use *two* quote marks where you want one to appear. For example, if you want to produce the text:

```
"Master Control"
```

in italics, you must type:

```
.i ""Master Control\|""
```

The `\|` produces a very narrow space so that the ‘|’ does not overlap the quote sign in TROFF, like this:

"*Master Control*"

There are also several "pseudo-fonts" available. The input:

```
.(b
.u underlined
.bi "bold italics"
.bx "words in a box"
.)b
```

generates

```
underlined
bold italics
words in a box
```

In NROFF these all just underline the text. Notice that pseudo font requests set only the single parameter in the pseudo font; ordinary font requests will begin setting all text in the special font if you do not provide a parameter. No more than one word should appear with these three font requests in the middle of lines. This is because of the way TROFF justifies text. For example, if you were to issue the requests:

```
.bi "some bold italics"
and
.bx "words in a box"
```

in the middle of a line TROFF would produce *some bold italics* and | words in a box], which I think you will agree does not look good.

The second parameter of all font requests is set in the original font. For example, the font request:

```
.b bold face
```

generates "bold" in bold font, but sets "face" in the font of the surrounding text, resulting in:

```
boldface.
```

To set the two words **bold** and **face** both in **bold face**, type:

```
.b "bold face"
```

You can mix fonts in a word by using the special sequence `\c` at the end of a line to indicate "continue text processing"; this allows input lines to be joined together without a space inbetween them. For example, the input:

```
.u under \c
.i italics
```

generates under*italics*, but if we had typed:

```
.u under
.i italics
```

the result would have been under *italics* as two words.

6.2. Point Sizes

The phototypesetter supports different sizes of type, measured in points. The default point size is 10 points for most text, 8 points for footnotes. To change the pointsize, type:

```
.sz +N
```

where *N* is the size wanted in points. The *vertical spacing* (distance between the bottom of most letters (the *baseline*) between adjacent lines) is set to be proportional to the type size.

Warning: changing point sizes on the phototypesetter is a slow mechanical operation. Size changes should be considered carefully.

6.3. Quotes

It is conventional when using the typesetter to use pairs of grave and acute accents to generate double quotes, rather than the double quote character (‘ ’). This is because it looks better to use grave and acute accents; for example, compare "quote" to "quote".

In order to make quotes compatible between the typesetter and terminals, you may use the sequences `*(lq` and `*(rq` to stand for the left and right quote respectively. These both appear as " on most terminals, but are typeset as " and " respectively. For example, use:

```
\*(lqSome things aren't true
even if they did happen.\*(rq
```

to generate the result:

```
"Some things aren't true even if they did happen."
```

As a shorthand, the special font request:

```
.q "quoted text"
```

will generate "quoted text". Notice that you must surround the material to be quoted with double quote marks if it is more than one word.

6.4. Special Characters

There are a number of special characters available. The escape sequences used to generate these characters are listed below.

Name	Usage	Example	
Acute accent	<code>*'´</code>	<code>a*'´</code>	á
Grave accent	<code>*'̀</code>	<code>e*'̀</code>	è
Umlat	<code>*:</code>	<code>u*:</code>	ü
Tilde	<code>*~</code>	<code>n*~</code>	ñ
Caret	<code>*^</code>	<code>e*^</code>	ê



Cedilla
Czech
Circle

/*,
/*v
/*o

c/*,
e/*v
A/*o

c
ç
ë
Ä





Appendix C: A Dictionary of vi Characters

This appendix explains the uses vi makes of every keyboard character. Characters are presented in their order in the ASCII character set: control characters come first, followed by the “special” characters, digits, uppercase, and lower-case characters.

For each character, this section explains the meaning it has when used as a command, as well as any meaning it has during an insert.

- ↑@ Not a command character. If typed as the first character of an insertion it is replaced with the last text inserted, and the insert terminates. Only 128 characters are saved from the last insert; if more characters were inserted the mechanism is not available. A ↑@ cannot be part of the file due to the editor implementation.
- ↑A Unused.
- ↑B Backward window. A count specifies repetition. Two lines of continuity are kept if possible.
- ↑C Unused.
- ↑D As a command, scrolls down a half-window of text. A count gives the number of (logical) lines to scroll, and is remembered for future ↑D and ↑U commands. During an insert, backtabs over *autoindent* white space at the beginning of a line. This white space cannot be backspaced over.
- ↑E Exposes one more line below the current screen in the file, leaving the cursor where it is if possible.
- ↑F Forward window. A count specifies repetition. Two lines of continuity are kept if possible.
- ↑G Equivalent to :fRETURN, printing the current file, whether it has been modified, the current line number and the number of lines in the file, and the percentage of the way through the file that you are.
- ↑H (BS) Same as **left arrow**. (See **h**). During an insert, eliminates the last input character, backing over it but not erasing it; it remains so you can see what you typed if you wish to type something only slightly different.
- ↑I (TAB) Not a command character. When inserted it prints as some number of spaces. When the cursor is at a tab character it rests at the last of the spaces which represent the tab. The spacing of tabstops is controlled by the *tabstop* option.
- ↑J (LF) Same as **down arrow** (see **j**).

- ↑K Unused.
- ↑L The ASCII formfeed character, this causes the screen to be cleared and redrawn. This is useful after a transmission error, if characters typed by a program other than the editor scramble the screen, or after output is stopped by an interrupt.
- ↑M (RETURN)
- A carriage return advances to the next line, at the first non-white position in the line. Given a count, it advances that many lines. During an insert, a RETURN causes the insert to continue onto another line.
- ↑N Same as **down arrow** (see **j**).
- ↑O Unused.
- ↑P Same as **up arrow** (see **k**).
- ↑Q Not a command character. In input mode, ↑Q quotes the next character, the same as ↑V, except that some teletype drivers will eat the ↑Q so that the editor never sees it.
- ↑R Redraws the current screen, eliminating logical lines not corresponding to physical lines (lines with only a single @ character on them). On hardcopy terminals in *open* mode, retypes the current line.
- ↑S Unused. Some teletype drivers use ↑S to suspend output until ↑Q is pressed.
- ↑T Not a command character. During an insert, with *autoindent* set and at the beginning of the line, inserts *shiftwidth* white space.
- ↑U Scrolls the screen up, inverting ↑D which scrolls down. Counts work as they do for ↑D, and the previous scroll amount is common to both. On a dumb terminal, ↑U will often necessitate clearing and redrawing the screen further back in the file.
- ↑V Not a command character. In input mode, quotes the next character so that it is possible to insert non-printing and special characters into the file.
- ↑W Not a command character. During an insert, backs up as **b** would in command mode; the deleted characters remain on the display (see ↑H).
- ↑X Unused.
- ↑Y Exposes one more line above the current screen, leaving the cursor where it is if possible. (No mnemonic value for this key; however, it is next to ↑U which scrolls up a bunch.)

- ↑Z If mapped to End-of-File (EOF), stops the editor, exiting to the top level shell. Same as **:stopRETURN**. Otherwise, unused.
- ↑[(ESC) Cancels a partially formed command, such as a **z** when no following character has yet been given; terminates inputs on the last line (read by commands such as **:** **/** and **?**); ends insertions of new text into the buffer. If an ESC is given when quiescent in command state, the editor rings the bell or flashes the screen. You can thus hit ESC if you don't know what is happening till the editor rings the bell. If you don't know if you are in insert mode you can type **ESCa**, and then material to be input; the material will be inserted correctly whether or not you were in insert mode when you started.
- ↑\ Unused.
- ↑} Searches for the word which is after the cursor as a tag. Equivalent to typing **:ta**, this word, and then a RETURN. Mnemonically, this command is "go right to".
- ↑↑ Equivalent to **:e #RETURN**, returning to the previous position in the last edited file, or editing a file which you specified if you got a 'No write since last change diagnostic' and do not want to have to type the file name again. (You have to do a **:w** before ↑↑ will work in this case. If you do not wish to write the file you should do **:e! #RETURN** instead.)
- ↑_ Unused. Reserved as the command character for the Tektronix 4025 and 4027 terminal.
- SPACE Same as **right arrow** (see l).
- ! An operator, which processes lines from the buffer with reformatting commands. Follow ! with the object to be processed, and then the command name terminated by RETURN. Doubling ! and preceding it by a count causes count lines to be filtered; otherwise the count is passed on to the object after the !. Thus **2!}fmtRETURN** reformats the next two paragraphs by running them through the program *fmt*. If you are working on LISP, the command **!%grindRETURN** given at the beginning of a function, will run the text of the function through the LISP grinder if it is present. To read a file or the output of a command into the buffer use **:r**. To simply execute a command use **!.**
- " Precedes a named buffer specification. There are named buffers **1-9** used for saving deleted text and named buffers **a-z** into which you can place text.
- # The macro character which, when followed by a number, will substitute for a function key on terminals without

- function keys. In input mode, if this is your erase character, it will delete the last character you typed in input mode, and must be preceded with a \ to insert it, since it normally backs over the last input character you gave.
- \$** Moves to the end of the current line. If you **:se listRETURN**, then the end of each line will be shown by printing a \$ after the end of the displayed text in the line. Given a count, advances to the count'th line following the next end-of-line; thus **2\$** advances to the end of the following line.
- %** Moves to the parenthesis or brace { } which balances the parenthesis or brace at the current cursor position.
- &** A synonym for **:&RETURN**, by analogy with the *ex &* command.
- '** When followed by a ' returns to the previous context at the beginning of a line. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter **a-z**, returns to the line which was marked with this letter with a **m** command, at the first non-white character in the line. When used with an operator such as **d**, the operation takes place over complete lines; if you use **`**, the operation takes place from the exact marked place to the current cursor position within the line.
- (** Retreats to the beginning of a sentence, or to the beginning of a LISP s-expression if the *lisp* option is set. A sentence ends at a **.!** or **?** which is followed by either the end of a line or by two spaces. Any number of closing **)] "** and **'** characters may appear after the **.!** or **?**, and before the spaces or end of line. Sentences also begin at paragraph and section boundaries (see **{** and **[[** below). A count advances that many sentences (4.2, 6.8).
-)** Advances to the beginning of a sentence. A count repeats the effect. See **(** above for the definition of a sentence (4.2, 6.8).
- *** Unused.
- +** Same as **RETURN** when used as a command.
- ,** Reverse of the last **f F t** or **T** command, looking the other way in the current line. Especially useful after hitting too many **;** characters. A count repeats the search.
- Retreats to the previous line at the first non-white character. This is the inverse of **+** and **RETURN**. If the line moved to is not on the screen, the screen is scrolled, or cleared and redrawn if this is not possible. If a large amount of scrolling would be required the screen is also

cleared and redrawn, with the current line at the center.

- . Repeats the last command which changed the buffer. Especially useful when deleting words or lines; you can delete some words/lines and then hit . to delete more and more words/lines. Given a count, it passes it on to the command being repeated. Thus after a **2dw**, **3.** deletes three words.
- / Reads a string from the last line on the screen, and scans forward for the next occurrence of this string. The normal input editing sequences may be used during the input on the bottom line; an returns to command state without ever searching. The search begins when you hit RETURN to terminate the pattern; the cursor moves to the beginning of the last line to indicate that the search is in progress; the search may then be terminated with a DEL or RUB, or by backspacing when at the beginning of the bottom line, returning the cursor to its initial position. Searches normally wrap end-around to find a string anywhere in the buffer.

When used with an operator the enclosed region is normally affected. By mentioning an offset from the line matched by the pattern you can force whole lines to be affected. To do this give a pattern with a closing / and then an offset $+n$ or $-n$.

To include the character / in the search string, you must escape it with a preceding \. A \uparrow at the beginning of the pattern forces the match to occur at the beginning of a line only; this speeds the search. A \$ at the end of the pattern forces the match to occur at the end of a line only. More extended pattern matching is available. Unless you set **nomagic** in your *.exrc* file you will have to precede the characters . [* and ~ in the search pattern with a \ to get them to work as you would otherwise expect.
- 0 Moves to the first character on the current line. Also used, in forming numbers, after an initial **1-9**.
- 1-9 Used to form numeric arguments to commands.
- : A prefix to a set of commands for file and option manipulation and escapes to the system. Input is given on the bottom line and terminated with an RETURN, and the command then executed. You can return to where you were by hitting DEL or RUB if you hit : accidentally.
- ; Repeats the last single character find which used **f F t** or **T**. A count iterates the basic scan.
- < An operator which shifts lines left one *shiftwidth*, normally 8 spaces. Like all operators, affects lines when repeated, as in <<. Counts are passed through to the basic object,

- thus **3<<** shifts three lines.
- =** Reindents line for LISP, as though they were typed in with *lisp* and *autoindent* set.
- >** An operator which shifts lines right one *shiftwidth*, normally 8 spaces. Affects lines when repeated as in **>>**. Counts repeat the basic object.
- ?** Scans backwards, the opposite of **/**. See the **/** description above for details on scanning.
- @** A macro character. If this is your kill character, you must escape it with a **** to type it in during input mode, as it normally backs over the input you have given on the current line.
- A** Appends at the end of line, a synonym for **\$a**.
- B** Backs up a word, where words are composed of non-blank sequences, placing the cursor at the beginning of the word. A count repeats the effect.
- C** Changes the rest of the text on the current line; a synonym for **c\$**.
- D** Deletes the rest of the text on the current line; a synonym for **d\$**.
- E** Moves forward to the end of a word, defined as blanks and non-blanks, like **B** and **W**. A count repeats the effect.
- F** Finds a single following character, backwards in the current line. A count repeats this search count times
- G** Goes to the line number given as preceding argument, or the end of the file if no preceding count is given. The screen is redrawn with the new current line in the center if necessary.
- H** Home arrow. Homes the cursor to the top line on the screen. If a count is given, then the cursor is moved to the count'th line on the screen. In any case the cursor is moved to the first non-white character on the line. If used as the target of an operator, full lines are affected.
- I** Inserts at the beginning of a line; a synonym for **↑i**.
- J** Joins together lines, supplying appropriate white space: one space between words, two spaces after a **.**, and no spaces at all if the first character of the joined on line is **)**. A count causes that many lines to be joined rather than the default two.
- K** Unused.

- L** Moves the cursor to the first non-white character of the last line on the screen. With a count, to the first non-white of the count'th line from the bottom. Operators affect whole lines when used with **L**.
- M** Moves the cursor to the middle line on the screen, at the first non-white position on the line.
- N** Scans for the next match of the last pattern given to / or ?, but in the reverse direction; this is the reverse of **n**.
- O** Opens a new line above the current line and inputs text there up to an ESC. A count can be used on dumb terminals to specify a number of lines to be opened; this is generally obsolete, as the *slowopen* option works better.
- P** Puts the last deleted text back before/above the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines. Otherwise the text is inserted between the characters before and at the cursor. May be preceded by a named buffer specification "x to retrieve the contents of the buffer; buffers 1-9 contain deleted material, buffers a-z are available for general use.
- Q** Quits from **vi** to *ex* command mode. In this mode, whole lines form commands, ending with a RETURN. You can give all the : commands; the editor supplies the : as a prompt.
- R** Replaces characters on the screen with characters you type (overlay fashion). Terminates with an ESC.
- S** Changes whole lines, a synonym for **cc**. A count substitutes for that many lines. The lines are saved in the numeric buffers, and erased on the screen before the substitution begins.
- T** Takes a single following character, locates the character before the cursor in the current line, and places the cursor just after that character. A count repeats the effect. Most useful with operators such as **d**.
- U** Restores the current line. Undoes any changes you have made.
- V** Unused.
- W** Moves forward to the beginning of a word in the current line, where words are defined as sequences of blank/non-blank characters. A count repeats the effect.
- X** Deletes the character before the cursor. A count repeats the effect, but only characters on the current line are deleted.

- Y** Yanks a copy of the current line into the unnamed buffer, to be put back by a later **p** or **P**; a very useful synonym for **yy**. A count yanks that many lines. May be preceded by a buffer name to put lines in that buffer.
- ZZ** Exits the editor. (Same as **:xRETURN**.) If any changes have been made, the buffer is written out to the current file. Then the editor quits.
- [[** Backs up to the previous section boundary. A section begins at each macro in the *sections* option, normally a **'NH'** or **'SH'** and also at lines which start with a formfeed **↑L**. Lines beginning with **{** also stop **[[**; this makes it useful for looking backwards, a function at a time, in C programs. If the option *lisp* is set, stops at each **(** at the beginning of a line, and is thus useful for moving backwards at the top level LISP objects.
- ** Unused.
-]]** Forward to a section boundary, see **[[** for a definition.
- ↑** Moves to the first non-white position on the current line.
- _** Unused.
- `** When followed by a **`** returns to the previous context. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter **a-z**, returns to the position which was marked with this letter with a **m** command. When used with an operator such as **d**, the operation takes place from the exact marked place to the current position within the line; if you use **'**, the operation takes place over complete lines.
- a** Appends arbitrary text after the current cursor position; the insert can continue onto multiple lines by using **RETURN** within the insert. A count causes the inserted text to be replicated, but only if the inserted text is all on one line. The insertion terminates with an **ESC** .
- b** Backs up to the beginning of a word in the current line. A word is a sequence of alphanumerics, or a sequence of special characters. A count repeats the effect.
- c** An operator which changes the following object, replacing it with the following input text up to an **ESC**. If more than part of a single line is affected, the text which is changed away is saved in the numeric named buffers. If only part of the current line is affected, then the last character to be changed away is marked with a **\$**. A count causes that many objects to be affected, thus both **3c**) and **c3**) change the following three sentences.

- d** An operator which deletes the following object. If more than part of a line is affected, the text is saved in the numeric buffers. A count causes that many objects to be affected; thus **3dw** is the same as **d3w**.
- e** Advances to the end of the next word, defined as for **b** and **w**. A count repeats the effect.
- f** Finds the first instance of the next character following the cursor on the current line. A count repeats the find.
- g** Unused.
- h** Left arrow. Moves the cursor one character to the left. Like the other arrow keys, either **h**, the **left arrow** key, or one of the synonyms (**↑H**) has the same effect. On v2 editors, arrow keys on certain kinds of terminals (those which send escape sequences, such as vt52, c100, or hp) cannot be used. A count repeats the effect.
- i** Inserts text before the cursor, otherwise like **a**.
- j** Down arrow. Moves the cursor one line down in the same column. If the position does not exist, *vi* comes as close as possible to the same column. Synonyms include **↑J** (linefeed) and **↑N**.
- k** Up arrow. Moves the cursor one line up. **↑P** is a synonym.
- l** Right arrow. Moves the cursor one character to the right. **SPACE** is a synonym.
- m** Marks the current position of the cursor in the mark register which is specified by the next character **a-z**. Return to this position or use with an operator using **`** or **'**.
- n** Repeats the last / or ? scanning commands.
- o** Opens new lines below the current line; otherwise like **O**.
- p** Puts text after/below the cursor; otherwise like **P**.
- q** Unused.
- r** Replaces the single character at the cursor with a single character you type. The new character may be a RETURN; this is the easiest way to split lines. A count replaces each of the following count characters with the single character given; see **R** above which is the more usually useful iteration of **r**.
- s** Changes the single character under the cursor to the text which follows up to an ESC; given a count, that many characters from the current line are changed. The last character to be changed is marked with **\$** as in **c**.

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

Appendix D: Writing Tools - The STYLE and DICTION Programs

L. L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

W. Vesterman

Livingston College
Rutgers University

1. Introduction

Computers have become important in the document preparation process, with programs to check for spelling errors and to format documents. As the amount of text stored on line increases, it becomes feasible and attractive to study writing style and to attempt to help the writer in producing readable documents. The system of writing tools described here is a first step toward such help. The system includes programs and a data base to analyze writing style at the word and sentence level. We use the term "style" in this paper to describe the results of a writer's particular choices among individual words and sentence forms. Although many judgements of style are subjective, particularly those of word choice, there are some objective measures that experts agree lead to good style. Three programs have been written to measure some of the objectively definable characteristics of writing style and to identify some commonly misused or unnecessary phrases. Although a document that conforms to the stylistic rules is not guaranteed to be coherent and readable, one that violates all of the rules is likely to be difficult or tedious to read. The program STYLE calculates readability, sentence length variability, sentence type, word usage and sentence openers at a rate of about 400 words per second on a PDP11/70 running the UNIX† Operating System. It assumes that the sentences are well-formed, i. e. that each sentence has a verb and that the subject and verb agree in number. DICTION identifies phrases that are either bad usage or unnecessarily wordy. EXPLAIN acts as a thesaurus for the phrases found by DICTION. Sections 2, 3, and 4 describe the programs; Section 5 gives the results on a cross-section of technical documents; Section 6 discusses accuracy and problems; Section 7 gives implementation details.

2. STYLE

The program STYLE reads a document and prints a summary of readability indices, sentence length and type, word usage, and sentence openers. It may also be used to locate all sentences in a document longer than a given length, of readability index higher than a given number, those containing a passive verb, or those beginning with an expletive. STYLE is based on the system for finding English word classes or parts of speech, PARTS [1]. PARTS is a set of programs that uses a small dictionary (about 350 words) and suffix rules to partially assign word classes to English text. It then uses experimentally derived rules of word order to assign word classes to all words in the text with an accuracy of about 95%. Because PARTS uses only a small dictionary and general rules, it works on text about any subject, from physics to psychology. Style measures have been built into the output phase of the programs that make up PARTS. Some of the measures are simple counters of the word classes found by PARTS; many are more complicated. For example, the verb count is the total number of verb phrases. This includes phrases like:

has been going
was only going
to go

each of which each counts as one verb. Figure 1 shows the output of STYLE run on a paper by Kernighan and Mashey about the UNIX programming environment [2]. As the example shows, STYLE output is in five parts. After a brief discussion of sentences, we will describe the parts in order.

2.1. What is a sentence?

Readers of documents have little trouble deciding where the sentences end. People don't even have to stop and think about uses of the character "." in constructions like 1.25, A. J. Jones, Ph.D., i. e., or etc. . When a computer reads a document, finding the end of sentences is not as easy. First we must throw away the printer's marks and formatting commands that litter the text in computer form. Then STYLE defines a sentence as a string of words ending in one of:

programming environment	
readability grades:	(Kincaid) 12.3 (auto) 12.8 (Coleman-Liau) 11.8 (Flesch) 13.5 (46.3)
sentence info:	no. sent 335 no. wds 7419 av sent leng 22.1 av word leng 4.91 no. questions 0 no. imperatives 0 no. nonfunc wds 4362 58.8% av leng 6.38 short sent (<17) 35% (118) long sent (>32) 16% (55) longest sent 82 wds at sent 174; shortest sent 1 wds at sent 117
sentence types:	simple 34% (114) complex 32% (108) compound 12% (41) compound-complex 21% (72)
word usage:	verb types as % of total verbs tobe 45% (373) aux 16% (133) inf 14% (114) passives as % of non-inf verbs 20% (144) types as % of total prep 10.8% (804) conj 3.5% (262) adv 4.8% (354) noun 26.7% (1983) adj 18.7% (1388) pron 5.3% (393) nominalizations 2 % (155)
sentence beginnings:	subject opener: noun (63) pron (43) pos (0) adj (58) art (62) tot 67% prep 12% (39) adv 9% (31) verb 0% (1) sub_conj 6% (20) conj 1% (5) expletives 4% (13)

Figure 1

.! ? /.

The end marker “/.” may be used to indicate an imperative sentence. Imperative sentences that are not so marked are not identified as imperative. STYLE properly handles numbers with embedded decimal points and commas, strings of letters and numbers with embedded decimal points used for naming computer file names, and the common abbreviations listed in Addendum 1. Numbers that end sentences, like the preceding sentence, cause a sentence break if the next word begins with a capital letter. Initials only cause a sentence break if the next word begins with a capital and is found in the dictionary of function words used by PARTS. So the string

J. D. JONES

does not cause a break, but the string

... system H. The ...

does. With these rules most sentences are broken at the proper place, although occasionally either two sentences are called one or a fragment is called a sentence. More on this later.

2.2. Readability Grades

The first section of STYLE output consists of four readability indices. As Klare points out in [3] readability indices may be used to estimate the reading skills needed by the reader to understand a document. The readability indices reported by STYLE are based on measures of sentence and word lengths. Although the indices may not measure whether the document is coherent and well organized, experience has shown that high indices seem to be indicators of stylistic difficulty. Documents with short sentences and short words have low scores; those with long sentences and many polysyllabic words have high scores. The 4 formulae reported are Kincaid Formula [4], Automated Readability Index [5], Coleman-Liau Formula [6] and a normalized version of Flesch Reading Ease Score [7]. The formulae differ because they were experimentally derived using different texts and subject groups. We will discuss each of the formulae briefly; for a more detailed discussion the reader should see [3].

The Kincaid Formula, given by:

$$Reading_Grade=11.8*syl_per_wd+.39*wds_per_sent-15.59$$

was based on Navy training manuals that ranged in difficulty from 5.5 to 16.3 in reading grade level. The score reported by this formula tends to be in the mid-range of the 4 scores. Because it is based on adult training manuals rather than school book text, this formula is probably the best one to apply to technical documents.

The Automated Readability Index (ARI), based on text from grades 0 to 7, was derived to be easy to automate. The formula is:

$$Reading_Grade=4.71*let_per_wd+.5*wds_per_sent-21.43$$

ARI tends to produce scores that are higher than Kincaid and Coleman-Liau but are usually slightly lower than Flesch.

The Coleman-Liau Formula, based on text ranging in difficulty from .4 to 16.3, is:

$$Reading_Grade=5.89*let_per_wd-.3*sent_per_100_wds-15.8$$

Of the four formulae this one usually gives the lowest grade when applied to technical documents.

The last formula, the Flesch Reading Ease Score, is based on grade school text covering grades 3 to 12. The formula, given by:

$$Reading_Score=206.835-84.6*syl_per_wd-1.015*wds_per_sent$$

is usually reported in the range 0 (very difficult) to 100 (very easy). The score reported by STYLE is scaled to be comparable to the other formulas, except that the maximum grade level reported is set to 17. The Flesch score is usually the highest of the 4 scores on technical documents.

Coke [8] found that the Kincaid Formula is probably the best predictor for technical documents; both ARI and Flesch tend to overestimate the difficulty; Coleman-Liau tend to underestimate. On text in the range of grades 7 to 9 the four formulas tend to be about the same. On easy text the Coleman-Liau formula is probably preferred since it is reasonably accurate at the lower grades and it is safer to present text that is a little too easy than a little too hard.

If a document has particularly difficult technical content, especially if it includes a lot of mathematics, it is probably best to make the text very easy to read, i.e. a lower readability index by shortening the sentences and words. This will allow the reader to concentrate on the technical content and not the long sentences. The user should remember that these indices are estimators; they should not be taken as absolute numbers. STYLE called with “-r number” will print all sentences with an Automated Readability Index equal to or greater than “number”.

2.3. Sentence length and structure

The next two sections of STYLE output deal with sentence length and structure. Almost all books on writing style or effective writing emphasize the importance of variety in sentence length and structure for good writing. Ewing's first rule in discussing style in the book *Writing for Results* [9] is:

“Vary the sentence structure and length of your sentences.”

Leggett, Mead and Charvat break this rule into 3 in *Prentice-Hall Handbook for Writers* [10] as follows:

“34a. Avoid the overuse of short simple sentences.”

“34b. Avoid the overuse of long compound sentences.”

“34c. Use various sentence structures to avoid monotony and increase effectiveness.”

Although experts agree that these rules are important, not all writers follow them. Sample technical documents have been found with almost no sentence length or type variability. One document had 90% of its sentences about the same length as the average; another was made up almost entirely of simple sentences (80%).

The output sections labeled “sentence info” and “sentence types” give both length and structure measures. STYLE reports on the number and average length of both sentences and words, and number of questions and imperative sentences (those ending in “/.”). The measures of non-function words are an attempt to look at the content words in the document. In English non-function words are nouns, adjectives, adverbs, and non-auxiliary verbs; function words are prepositions, conjunctions, articles, and auxiliary verbs. Since most function words are short, they tend to lower the average word length. The average length of non-function words may be a more useful measure for comparing word choice of different writers than the total average word length. The percentages of short and long sentences measure sentence length variability. Short sentences are those at least 5 words less than the average; long sentences are those at least 10 words longer than the average. Last in the sentence information section is the length and location of the longest and shortest sentences. If the flag “-l number” is used, STYLE will print all sentences longer than “number”.

Because of the difficulties in dealing with the many uses of commas and conjunctions in English, sentence type definitions vary slightly from those of standard textbooks, but still measure the same constructional activity.

1. A simple sentence has one verb and no dependent clause.

2. A complex sentence has one independent clause and one dependent clause, each with one verb. Complex sentences are found by identifying sentences that contain either a subordinate conjunction or a clause beginning with words like "that" or "who". The preceding sentence has such a clause.
3. A compound sentence has more than one verb and no dependent clause. Sentences joined by ";" are also counted as compound.
4. A compound-complex sentence has either several dependent clauses or one dependent clause and a compound verb in either the dependent or independent clause.

Even using these broader definitions, simple sentences dominate many of the technical documents that have been tested, but the example in Figure 1 shows variety in both sentence structure and sentence length.

2.4. Word Usage

The word usage measures are an attempt to identify some other constructional features of writing style. There are many different ways in English to say the same thing. The constructions differ from one another in the form of the words used. The following sentences all convey approximately the same meaning but differ in word usage:

The cxio program is used to perform all communication between the systems.

The cxio program performs all communications between the systems.

The cxio program is used to communicate between the systems.

The cxio program communicates between the systems.

All communication between the systems is performed by the cxio program.

The distribution of the parts of speech and verb constructions helps identify overuse of particular constructions. Although the measures used by STYLE are crude, they do point out problem areas. For each category, STYLE reports a percentage and a raw count. In addition to looking at the percentage, the user may find it useful to compare the raw count with the number of sentences. If, for example, the number of infinitives is almost equal to the number of sentences, then many of the sentences in the document are constructed like the first and third in the preceding example. The user may want to transform some of these sentences into another form. Some of the implications of the word usage measures are discussed below.

Verbs are measured in several different ways to try to determine what types of verb constructions are most frequent in the document. Technical writing tends to contain many passive verb constructions and other usage of the verb "to be". The category of verbs labeled "tobe" measures both passives and sentences of the form:

subject tobe predicate

In counting verbs, whole verb phrases are counted as one verb. Verb phrases containing auxiliary verbs are counted in the category "aux". The verb phrases counted here are those whose tense is not simple present or simple past. It might eventually be useful to do more detailed measures of verb tense or mood. Infinitives are listed as "inf". The percentages reported for these three categories are based on the total number of verb phrases found. These categories are not mutually exclusive; they cannot be added, since, for

example, "to be going" counts as both "tobe" and "inf". Use of these three types of verb constructions varies significantly among authors.

STYLE reports passive verbs as a percentage of the finite verbs in the document. Most style books warn against the overuse of passive verbs. Coleman [11] has shown that sentences with active verbs are easier to learn than those with passive verbs. Although the inverted object-subject order of the passive voice seems to emphasize the object, Coleman's experiments showed that there is little difference in retention by word position. He also showed that the direct object of an active verb is retained better than the subject of a passive verb. These experiments support the advice of the style books suggesting that writers should try to use active verbs wherever possible. The flag "-p" causes STYLE to print all sentences containing passive verbs.

Pronouns add cohesiveness and connectivity to a document by providing back-reference. They are often a short-hand notation for something previously mentioned, and therefore connect the sentence containing the pronoun with the word to which the pronoun refers. Although there are other mechanisms for such connections, documents with no pronouns tend to be wordy and to have little connectivity.

Adverbs can provide transition between sentences and order in time and space. In performing these functions, adverbs, like pronouns, provide connectivity and cohesiveness.

Conjunctions provide parallelism in a document by connecting two or more equal units. These units may be whole sentences, verb phrases, nouns, adjectives, or prepositional phrases. The compound and compound-complex sentences reported under sentence type are parallel structures. Other uses of parallel structures are indicated by the degree that the number of conjunctions reported under word usage exceeds the compound sentence measures.

Nouns and Adjectives. A ratio of nouns to adjectives near unity may indicate the over-use of modifiers. Some technical writers qualify every noun with one or more adjectives. Qualifiers in phrases like "simple linear single-link network model" often lend more obscurity than precision to a text.

Nominalizations are verbs that are changed to nouns by adding one of the suffixes "ment", "ance", "ence", or "ion". Examples are accomplishment, admittance, adherence, and abbreviation. When a writer transforms a nominalized sentence to a non-nominalized sentence, she/he increases the effectiveness of the sentence in several ways. The noun becomes an active verb and frequently one complicated clause becomes two shorter clauses. For example,

Their inclusion of this provision is admission of the importance of the system.

When they included this provision, they admitted the importance of the system.

Coleman found that the transformed sentences were easier to learn, even when the transformation produced sentences that were slightly longer, provided the transformation broke one clause into two. Writers who find their document contains many nominalizations may want to transform some of the sentences to use active verbs.

2.5. Sentence openers

Another agreed upon principle of style is variety in sentence openers. Because STYLE determines the type of sentence opener by looking at the part of speech of the first word in the sentence, the sentences counted under the heading "subject opener" may not all really begin with the subject. However, a large percentage of sentences in this category still indicates lack of variety in sentence openers. Other sentence opener measures help the user determine if there are transitions between sentences and where the subordination occurs. Adverbs and conjunctions at the beginning of sentences are mechanisms for transition between sentences. A pronoun at the beginning shows a link to something previously mentioned and indicates connectivity.

The location of subordination can be determined by comparing the number of sentences that begin with a subordinator with the number of sentences with complex clauses. If few sentences start with subordinate conjunctions then the subordination is embedded or at the end of the complex sentences. For variety the writer may want to transform some sentences to have leading subordination.

The last category of openers, expletives, is commonly overworked in technical writing. Expletives are the words "it" and "there", usually with the verb "to be", in constructions where the subject follows the verb. For example,

There are three streets used by the traffic.
There are too many users on this system.

This construction tends to emphasize the object rather than the subject of the sentence. The flag "-e" will cause STYLE to print all sentences that begin with an expletive.

3. DICTION

The program DICTION prints all sentences in a document containing phrases that are either frequently misused or indicate wordiness. The program, an extension of Aho's FGREP [12] string matching program, takes as input a file of phrases or patterns to be matched and a file of text to be searched. A data base of about 450 phrases has been compiled as a default pattern file for DICTION. Before attempting to locate phrases, the program maps upper case letters to lower case and substitutes blanks for punctuation. Sentence boundaries were deemed less critical in DICTION than in STYLE, so abbreviations and other uses of the character "." are not treated specially. DICTION brackets all pattern matches in a sentence with the characters "[" "]" . Although many of the phrases in the default data base are correct in some contexts, in others they indicate wordiness. Some examples of the phrases and suggested alternatives are:

Phrase	Alternative
a large number of	many
arrive at a decision	decide
collect together	collect
for this reason	so
pertaining to	about
through the use of	by or with
utilize	use
with the exception of	except

Addendum 2 contains a complete list of the default file. Some of the entries are

short forms of problem phrases. For example, the phrase "the fact" is found in all of the following and is sufficient to point out the wordiness to the user:

Phrase	Alternative
accounted for by the fact that	caused by
an example of this is the fact that	thus
based on the fact that	because
despite the fact that	although
due to the fact that	because
in light of the fact that	because
in view of the fact that	since
notwithstanding the fact that	although

Entries in Addendum 2 preceded by "~" are not matched. See Section 7 for details on the use of "~".

The user may supply her/his own pattern file with the flag "-f patfile". In this case the default file will be loaded first, followed by the user file. This mechanism allows users to suppress patterns contained in the default file or to include their own pet peeves that are not in the default file. The flag "-n" will exclude the default file altogether. In constructing a pattern file, blanks should be used before and after each phrase to avoid matching substrings in words. For example, to find all occurrences of the word "the", the pattern " the " should be used. The blanks cause only the word "the" to be matched and not the string "the" in words like there, other, and therefore. One side effect of surrounding the words with blanks is that when two phrases occur without intervening words, only the first will be matched.

4. EXPLAIN

The last program, EXPLAIN, is an interactive thesaurus for phrases found by DICTION. The user types one of the phrases bracketed by DICTION and EXPLAIN responds with suggested substitutions for the phrase that will improve the diction of the document.

5. Results

5.1. STYLE

To get baseline statistics and check the program's accuracy, we ran STYLE on 20 technical documents. There were a total of 3287 sentences in the sample. The shortest document was 67 sentences long; the longest 339 sentences. The documents covered a wide range of subject matter, including theoretical computing, physics, psychology, engineering, and affirmative action. Table 1 gives the range, median, and standard deviation of the various style measures. As you will note most of the measurements have a fairly wide range of values across the sample documents.

As a comparison, Table 2 gives the median results for two different technical authors, a sample of instructional material, and a sample of the Federalist Papers. The two authors show similar styles, although author 2 uses somewhat shorter sentences and longer words than author 1. Author 1 uses all types of sentences, while author 2 prefers simple and complex sentences, using few compound or compound-complex sentences. The other major difference in the styles of these authors is the location of subordination. Author 1 seems to prefer embedded or

Table 1
Text Statistics on 20 Technical Documents

	variable	minimum	maximum	mean	standard deviation
Readability	Kincaid	9.5	16.9	13.3	2.2
	automated	9.0	17.4	13.3	2.5
	Cole-Liau	10.0	16.0	12.7	1.8
	Flesch	8.9	17.0	14.4	2.2
sentence info.	av sent length	15.5	30.3	21.6	4.0
	av word length	4.61	5.63	5.08	.29
	av nonfunction length	5.72	7.30	6.52	.45
	short sent	23%	46%	33%	5.9
	long sent	7%	20%	14%	2.9
sentence types	simple	31%	71%	49%	11.4
	complex	19%	50%	33%	8.3
	compound	2%	14%	7%	3.3
	compound-complex	2%	19%	10%	4.8
verb types	tobe	26%	64%	44.7%	10.3
	auxiliary	10%	40%	21%	8.7
	infinitives	8%	24%	15.1%	4.8
	passives	12%	50%	29%	9.3
word usage	prepositions	10.1%	15.0%	12.3%	1.6
	conjunction	1.8%	4.8%	3.4%	.9
	adverbs	1.2%	5.0%	3.4%	1.0
	nouns	23.6%	31.6%	27.8%	1.7
	adjectives	15.4%	27.1%	21.1%	3.4
	pronouns	1.2%	8.4%	2.5%	1.1
	nominalizations	2%	5%	3.3%	.8
sentence openers	prepositions	6%	19%	12%	3.4
	adverbs	0%	20%	9%	4.6
	subject	56%	85%	70%	8.0
	verbs	0%	4%	1%	1.0
	subordinating conj	1%	12%	5%	2.7
	conjunctions	0%	4%	0%	1.5
	expletives	0%	6%	2%	1.7

trailing subordination, while author 2 begins many sentences with the subordinate clause. The documents tested for both authors 1 and 2 were technical documents, written for a technical audience. The instructional documents, which are written for craftspeople, vary surprisingly little from the two technical samples. The sentences and words are a little longer, and they contain many passive and auxiliary verbs, few adverbs, and almost no pronouns. The instructional documents contain many imperative sentences, so there are many sentence with verb openers. The sample of Federalist Papers contrasts with the other samples in almost every way.

5.2. DICTION

In the few weeks that DICTION has been available to users about 35,000 sentences have been run with about 5,000 string matches. The authors using the program seem to make the suggested changes about 50-75% of the time. To date, almost 200 of the 450 strings in the default file have been matched. Although most of these phrases are valid and correct in some contexts, the 50-75% change rate seems to show that the phrases are used much more often than

Table 2
Text Statistics on Single Authors

	variable	author 1	author 2	inst.	FED
readability	Kincaid	11.0	10.3	10.8	16.3
	automated	11.0	10.3	11.9	17.8
	Coleman-Liau	9.3	10.1	10.2	12.3
	Flesch	10.3	10.7	10.1	15.0
sentence info	av sent length	22.64	19.61	22.78	31.85
	av word length	4.47	4.66	4.65	4.95
	av nonfunction length	5.64	5.92	6.04	6.87
	short sent	35%	43%	35%	40%
	long sent	18%	15%	16%	21%
sentence types	simple	36%	43%	40%	31%
	complex	34%	41%	37%	34%
	compound	13%	7%	4%	10%
	compound-complex	16%	8%	14%	25%
verb type	tobe	42%	43%	45%	37%
	auxiliary	17%	19%	32%	32%
	infinitives	17%	15%	12%	21%
	passives	20%	19%	36%	20%
word usage	prepositions	10.0%	10.8%	12.3%	15.9%
	conjunctions	3.2%	2.4%	3.9%	3.4%
	adverbs	5.05%	4.6%	3.5%	3.7%
	nouns	27.7%	26.5%	29.1%	24.9%
	adjectives	17.0%	19.0%	15.4%	12.4%
	pronouns	5.3%	4.3%	2.1%	6.5%
	nominalizations	1%	2%	2%	3%
sentence openers	prepositions	11%	14%	6%	5%
	adverbs	9%	9%	6%	4%
	subject	65%	59%	54%	66%
	verb	3%	2%	14%	2%
	subordinating conj	8%	14%	11%	3%
	conjunction	1%	0%	0%	3%
	expletives	3%	3%	0%	3%

concise diction warrants.

6. Accuracy

6.1. Sentence Identification

The correctness of the STYLE output on the 20 document sample was checked in detail. STYLE misidentified 129 sentence fragments as sentences and incorrectly joined two or more sentences 75 times in the 3287 sentence sample. The problems were usually because of nonstandard formatting commands, unknown abbreviations, or lists of non-sentences. An impossibly long sentence found as the longest sentence in the document usually is the result of a long list of non-sentences.

6.2. Sentence Types

Style correctly identified sentence type on 86.5% of the sentences in the sample. The type distribution of the sentences was 52.5% simple, 29.9% complex, 8.5% compound and 9% compound-complex. The program reported 49.5% simple, 31.9% complex, 8% compound and 10.4% compound-complex. Looking at the errors on the individual documents, the number of simple sentences was under-reported by about 4% and the complex and compound-complex were over-reported by 3% and 2%, respectively. The following matrix shows the programs output vs. the actual sentence type.

		Program Results			
		simple	complex	compound	comp-complex
Actual Sentence Type	simple	1566	132	49	17
	complex	47	892	6	65
	compound	40	6	207	23
	comp-complex	0	52	5	249

The system's inability to find imperative sentences seems to have little effect on most of the style statistics. A document with half of its sentences imperative was run, with and without the imperative end marker. The results were identical except for the expected errors of not finding verbs as sentence openers, not counting the imperative sentences, and a slight difference (1%) in the number of nouns and adjectives reported.

6.3. Word Usage

The accuracy of identifying word types reflects that of PARTS, which is about 95% correct. The largest source of confusion is between nouns and adjectives. The verb counts were checked on about 20 sentences from each document and found to be about 98% correct.

7. Technical Details

7.1. Finding Sentences

The formatting commands embedded in the text increase the difficulty of finding sentences. Not all text in a document is in sentence form; there are headings, tables, equations and lists, for example. Headings like "Finding Sentences" above should be discarded, not attached to the next sentence. However, since many of the documents are formatted to be phototypeset, and contain font changes, which usually operate on the most important words in the document, discarding all formatting commands is not correct. To improve the programs' ability to find sentence boundaries, the deformatting program, DEROFF [13], has been given some knowledge of the formatting packages used on the UNIX operating system. DEROFF will now do the following:

1. Suppress all formatting macros that are used for titles, headings, author's name, etc.
2. Suppress the arguments to the macros for titles, headings, author's name, etc.

3. Suppress displays, tables, footnotes and text that is centered or in no-fill mode.
4. Substitute a place holder for equations and check for hidden end markers. The place holder is necessary because many typists and authors use the equation setter to change fonts on important words. For this reason, header files containing the definition of the EQN delimiters must also be included as input to STYLE. End markers are often hidden when an equation ends a sentence and the period is typed inside the EQN delimiters.
5. Add a "." after lists. If the flag -ml is also used, all lists are suppressed. This is a separate flag because of the variety of ways the list macros are used. Often, lists are sentences that should be included in the analysis. The user must determine how lists are used in the document to be analyzed.

Both STYLE and DICTION call DEROFF before they look at the text. The user should supply the -ml flag if the document contains many lists of non-sentences that should be skipped.

7.2. Details of DICTION

The program DICTION is based on the string matching program FGREP. FGREP takes as input a file of patterns to be matched and a file to be searched and outputs each line that contains any of the patterns with no indication of which pattern was matched. The following changes have been added to FGREP:

1. The basic unit that DICTION operates on is a sentence rather than a line. Each sentence that contains one of the patterns is output.
2. Upper case letters are mapped to lower case.
3. Punctuation is replaced by blanks.
4. All pattern matches in the sentence are found and surrounded with "[" "]" .
5. A method for suppressing a string match has been added. Any pattern that begins with "~ " will not be matched. Because the matching algorithm finds the longest substring, the suppression of a match allows words in some correct contexts not to be matched while allowing the word in another context to be found. For example, the word "which" is often incorrectly used instead of "that" in restrictive clauses. However, "which" is usually correct when preceded by a preposition or ",". The default pattern file suppresses the match of the common prepositions or a double blank followed by "which" and therefore matches only the suspect uses. The double blank accounts for the replaced comma.

8. Conclusions

A system of writing tools that measure some of the objective characteristics of writing style has been developed. The tools are sufficiently general that they may be applied to documents on any subject with equal accuracy. Although the measurements are only of the surface structure of the text, they do point out problem areas. In addition to helping writers produce better documents, these programs may be useful for studying the writing process and finding other formulae for measuring readability.

References

1. L. L. Cherry, "PARTS - A System for Assigning Word Classes to English Text," submitted *Communications of the ACM*.
2. B. W. Kernighan and J. R. Mashey, "The UNIX Programming Environment," *Software - Practice & Experience*, **9**, 1-15 (1979).
3. G. R. Klare, "Assessing Readability," *Reading Research Quarterly*, 1974-1975, **10**, 62-102.
4. E. A. Smith and P. Kincaid, "Derivation and validation of the automated readability index for use with technical materials," *Human Factors*, 1970, **12**, 457-464.
5. J. P. Kincaid, R. P. Fishburne, R. L. Rogers, and B. S. Chissom, "Derivation of new readability formulas (Automated Readability Index, Fog count, and Flesch Reading Ease Formula) for Navy enlisted personnel," Navy Training Command Research Branch Report 8-75, Feb., 1975.
6. M. Coleman and T. L. Liau, "A Computer Readability Formula Designed for Machine Scoring," *Journal of Applied Psychology*, 1975, **60**, 283-284.
7. R. Flesch, "A New Readability Yardstick," *Journal of Applied Psychology*, 1948, **32**, 221-233.
8. E. U. Coke, private communication.
9. D. W. Ewing, *Writing for Results*, John Wiley & Sons, Inc., New York, N. Y. (1974).
10. G. Leggett, C. D. Mead and W. Charvat, *Prentice-Hall Handbook for Writers*, Seventh Edition, Prentice-Hall Inc., Englewood Cliffs, N. J. (1978).
11. E. B. Coleman, "Learning of Prose Written in Four Grammatical Transformations," *Journal of Applied Psychology*, 1965, vol. 49, no. 5, pp. 332-341.
12. A. V. Aho and M. J. Corasick, "Efficient String Matching: an aid to Bibliographic Search," *Communications of the ACM*, **18**, (6), 333-340, June 1975.
13. Bell Laboratories, "UNIX TIME-SHARING SYSTEM: UNIX PROGRAMMER'S MANUAL," Seventh Edition, Vol. 1 (January 1979).

Addendum 1

STYLE Abbreviations

a. d.
A. M.
a. m.
b. c.
Ch.
ch.
ckts.
dB.
Dept.
dept.
Depts.
depts.
Dr.
Drs.
e. g.
Eq.
eq.
et al.
etc.
Fig.
fig.
Figs.
figs.
ft.
i. e.
in.
Inc.
Jr.
jr.
mi.
Mr.
Mrs.
Ms.
No.
no.
Nos.
nos.
P. M.
p. m.
Ph. D.
Ph. d.
Ref.
ref.
Refs.
refs.

St.
vs.
yr.



Addendum 2

Default DICTION Patterns

a great deal of	center portion	fearful that	in the form of
a large number of	check into	few in number	in the instance of
a lot of	check on	file away	in the interim
a majority of	check up on	final completion	in the last analysis
a need for	circle around	final ending	in the matter of
a number of	close proximity	final outcome	in the near future
a particular preference for	collaborate together	final result	in the neighborhood of
a preference for	collect together	finalize	in the not too distant future
a small number of	combine together	find it interesting to know	in the proximity of
a tendency to	come to an end	first and foremost	in the range of
abovementioned	commence	first beginnings	in the same way as described
absolutely complete	common accord	first initiated	in the shape of
absolutely essential	compensation	firstly	in the vicinity of
accomplished	completely eliminated	follow after	in this case
accordingly	comprise	following after	in view of the
activate	concerning	for the purpose of	in violation of
actual	conduct an investigation of	for the reason that	inasmuch as
added increments	conjecture	for the simple reason that	indicate
adequate enough	connect up	for this reason	indicative of
advent	consensus of opinion	for your information	initialize
afford an opportunity	consequent result	from the point of view of	initiate
aggregate	consolidate together	full and complete	injurious to
all of	construct	generally agreed	inquire
all throughout	contemplate	good and	inside of
along the line	continue on	got to	institute a
an indication of	continue to remain	gratuitous	intents and purposes
analyzation	could of	greatly minimize	intermingle
and etc	count up	head up	irregardless
and or	couple together	help but	is defined as
another additional	debate about	helps in the production of	is used to control
any and all	decide on	hopeful	is when
arrive at a	deleterious effect	if and when	is where
as a matter of fact	demean	if at all possible	it is incumbent
as a method of	demonstrate	impact	it stands to reason
as good or better than	depreciate in value	implement	it was noted that if
as of now	deserving of	important essentials	joint cooperation
as per	desirable benefits	importantly	joint partnership
as regards	desirous of	in a large measure	just exactly
as related to	different than	in a position to	kind of
as to	discontinue	in accordance	know about
assistance	disutility	in advance of	last but not least
assistance to	divide up	in agreement with	later on
assistance to	doubt but	in all cases	leaving out of consideration
assuming that	due to	in back of	liable
at a later date	duly noted	in behalf of	link up
at about	during the time that	in behind	literally
at above	each and every	in between	little doubt that
at all times	early beginnings	in case	lose out on
at an early date	effectuate	in close proximity	lots of
at below	emotional feelings	in conflict with	main essentials
at the present	empty out	in conjunction with	make a
at the time when	enclosed herein	in connection with	make adjustments to
at this point in time	enclosed herewith	in fact	make an
at this time	end result	in large measure	make application to
at which time	end up	in many cases	make contact with
at your earliest convenience	endeavor	in most cases	make mention of
authorization	enter in	in my opinion I think	make out a list of
awful	enter into	in order to	make the acquaintance of
basic fundamentals	enthused	in rare cases	make the adjustment
basically	entirely complete	in reference to	manner
be cognizant of	equally good as	in regard to	maximum possible
being as	essentially	in regards to	meaningful
being that	eventuate	in relation with	meet up with
brief in duration	every now and then	in short supply	melt down
bring to a conclusion	exactly identical	in size	melt up
but that	experiencing difficulty	in terms of	methodology
but what	fabricate	in the amount of	might of
by means of	face up to	in the case of	minimize as far as possible
by the use of	facilitate	in the course of	minor importance
carry out experiments	facts and figures	in the event	miss out on
center about	fast in action	in the field of	modification
center around	fearful of		

more preferable	seems apparent	worth while
most unique	send a communication	would of
must of	short space of time	ing behavior
mutual cooperation	should of	wise
necessary requisite	single unit	- which
necessitate	situation	- about which
need for	so as to	- after which
nice	sort of	- at which
not be un	spell out	- between which
not in a position to	still continue	- by which
not of a high order of accuracy	still remain	- for which
not un	subsequent	- from which
notwithstanding	substantially in agreement	- in which
of considerable magnitude	succeed in	- into which
of that	suggestive of	- of which
of the opinion that	superior than	- on which
off of	surrounding circumstances	- on which
on a few occasions	take appropriate	- over which
on account of	take cognizance of	- through which
on behalf of	take into consideration	- to which
on the grounds that	termed as	- under which
on the occasion	terminate	- upon which
on the part of	termination	- with which
one of the	the author	- without which
open up	the authors	- clockwise
operates to correct	the case that	- likewise
outside of	the fact	- otherwise
over with	the foregoing	
overall	the foreseeable future	
past history	the fullest possible extent	
perceptive of	the majority of	
perform a measurement	the nature	
perform the measurement	the necessity of	
permits the reduction of	the only difference being that	
personalize	the order of	
pertaining to	the point that	
physical size	the truth is	
plan ahead	there are not many	
plan for the future	through the medium of	
plan in advance	through the use of	
plan on	throughout the entire	
present a conclusion	time interval	
present a report	to summarize the above	
presently	total effect of all this	
prior to	totality	
prioritize	transpire	
proceed to	true facts	
procure	try and	
productive of	ultimate end	
prolong the duration	under a separate cover	
protrude out from	under date of	
provided that	under separate cover	
pursuant to	under the necessity to	
put to use in	underlying purpose	
range all the way from	undertake a study	
reason is because	uniformly consistent	
reason why	unique	
recur again	until such time as	
reduce down	up to this time	
refer back	upshot	
reference to this	utilize	
reflective of	very	
regarding	very complete	
regretful	very unique	
reinitiate	vital	
relative to	which	
repeat again	with a view to	
representative of	with reference to	
resultant effect	with regard to	
resume again	with the exception of	
retreat back	with the object of	
return again	with the result that	
return back	with this in mind, it is clear that	
revert back	within the realm of possibility	
seal off	without further delay	

Appendix E : Refer — A Bibliography System

Bill Tuthill

Computing Services
University of California
Berkeley, CA 94720

Introduction

Taken together, the **refer** programs constitute a database system for use with variable-length information. To distinguish various types of bibliographic material, the system uses labels composed of upper case letters, preceded by a percent sign and followed by a space. For example, one document might be given this entry:

```
%A  Joel Kies  
%T  Document Formatting on Unix Using the -ms Macros  
%I  Computing Services  
%C  Berkeley  
%D  1980
```

Each line is called a field, and lines grouped together are called a record; records are separated from each other by a blank line. Bibliographic information follows the labels, containing data to be used by the **refer** system. The order of fields is not important, except that authors should be entered in the same order as they are listed on the document. Fields can be as long as necessary, and may even be continued on the following line(s).

The labels are meaningful to **nroff/troff** macros, and, with a few exceptions, the **refer** program itself does not pay attention to them. This implies that you can change the label codes, if you also change the macros used by **nroff/troff**. The macro package takes care of details like proper ordering, underlining the book title or journal name, and quoting the article's title. Here are the labels used by **refer**, with an indication of what they represent:

%H Header commentary, printed before reference
 %A Author's name
 %Q Corporate or foreign author (unreversed)
 %T Title of article or book
 %S Series title
 %J Journal containing article
 %B Book containing article
 %R Report, paper, or thesis (for unpublished material)
 %V Volume
 %N Number within volume
 %E Editor of book containing article
 %P Page number(s)
 %I Issuer (publisher)
 %C City where published
 %D Date of publication
 %O Other commentary, printed at end of reference
 %K Keywords used to locate reference
 %L Label used by -k option of **refer**
 %X Abstract (used by **roffbib**, not by **refer**)

Only relevant fields should be supplied. Except for %A, each field should be given only once; in the case of multiple authors, the senior author should come first. The %Q is for organizational authors, or authors with Japanese or Arabic names, in which cases the order of names should be preserved. Books should be labeled with the %T, not with the %B, which is reserved for books containing articles. The %J and %B fields should never appear together, although if they do, the %J will override the %B. If there is no author, just an editor, it is best to type the editor in the %A field, as in this example:

%A Bertrand Bronson, ed.

The %E field is used for the editor of a book (%B) containing an article, which has its own author. For unpublished material such as theses, use the %R field; the title in the %T field will be quoted, but the contents of the %R field will not be underlined. Unlike other fields, %H, %O, and %X should contain their own punctuation. Here is a modest example:

%A Mike E. Lesk
 %T Some Applications of Inverted Indexes on the Unix System
 %B Unix Programmer's Manual
 %I Bell Laboratories
 %C Murray Hill, NJ
 %D 1978
 %V 2a
 %K refer mkey inv hunt
 %X Difficult to read paper that dwells on indexing strategies,
 giving little practical advice about using \fRefer\fP.

Note that the author's name is given in normal order, without inverting the surname; inversion is done automatically, except when %Q is used instead of %A. We use %X rather than %O for the commentary because we do not want the comment printed all the time. The %O and %H fields are printed by both **refer** and **roffbib**; the %X field is printed only by **roffbib**, as a detached annotation paragraph.

Data Entry with Addbib

The **addbib** program is for creating and extending bibliographic databases. You must give it the filename of your bibliography:

% addbib database

Every time you enter **addbib**, it asks if you want instructions. To get them, type **y**; to skip them, type RETURN. **Addbib** prompts for various fields, reads from the keyboard, and writes records containing the **refer** codes to the database. After finishing a field entry, you should end it by typing RETURN. If a field is too long to fit on a line, type a backslash (\) at the end of the line, and you will be able to continue on the following line. Note: the backslash works in this capacity only inside **addbib**.

A field will not be written to the database if nothing is entered into it. Typing a minus sign as the first character of any field will cause **addbib** to back up one field at a time. Backing up is the best way to add multiple authors, and it really helps if you forget to add something important. Fields not contained in the prompting skeleton may be entered by typing a backslash as the last character before RETURN. The following line will be sent verbatim to the database and **addbib** will resume with the next field. This is identical to the procedure for dealing with long fields, but with new fields, don't forget the **%** key-letter.

Finally, you will be asked for an abstract (or annotation), which will be preserved as the **%X** field. Type in as many lines as you need, and end with a control-D (hold down the CTRL button, then press the d key). This prompting for an abstract can be suppressed with the **-a** command line option.

After one bibliographic record has been completed, **addbib** will ask if you want to continue. If you do, type RETURN; to quit, type **q** or **n** (quit or no). It is also possible to use one of the system editors to correct mistakes made while entering data. After the Continue? prompt, type any of the following: **edit**, **ex**, **vi**, or **ed** you will be placed inside the corresponding editor, and returned to **addbib** afterwards, from where you can either quit or add more data.

If the prompts normally supplied by **addbib** are not enough, are in the wrong order, or are too numerous, you can redefine the skeleton by constructing a promptfile. Create some file, to be named after the **-p** command line option. Place the prompts you want on the left side, followed by a single TAB (control-I), then the **refer** code that is to appear in the bibliographic database. **Addbib** will send the left side to the screen, and the right side, along with data entered, to the database.

Printing the Bibliography

Sortbib is for sorting the bibliography by author (**%A**) and date (**%D**), or by data in other fields. It is quite useful for producing bibliographies and annotated bibliographies, which are seldom entered in strict alphabetical order. It takes as arguments the names of up to 16 bibliography files, and sends the sorted records to standard output (the terminal screen), which may be redirected through a pipe or into a file.

The **-sKEYS** flag to **sortbib** will sort by fields whose key-letters are in the **KEYS** string, rather than merely by author and date. Key-letters in **KEYS** may be followed by a '+' to indicate that all such fields are to be used. The default is to sort by senior author and date (printing the senior author last name first), but **-sA+D** will sort by all authors and then date, and **-sATD** will sort on senior author, then title, and then date.

Roffbib is for running off the (probably sorted) bibliography. It can handle annotated bibliographies annotations are entered in the %X (abstract) field. **Roffbib** is a shell script that calls **refer -B** and **nroff -mbib**. It uses the macro definitions that reside in /usr/lib/tmac/tmac.bib, which you can redefine if you know **nroff** and **troff**. Note that **refer** will print the %H and %O commentaries, but will ignore abstracts in the %X field; **roffbib** will print both fields, unless annotations are suppressed with the -x option.

The following command sequence will lineprint the entire bibliography, organized alphabetically by author and date:

```
% sortbib database | roffbib | lpr
```

This is a good way to proofread the bibliography, or to produce a stand-alone bibliography at the end of a paper. Incidentally, **roffbib** accepts all flags used with **nroff**. For example:

```
% sortbib database | roffbib -Tdtc -s1
```

will make accent marks work on a DTC daisy-wheel printer, and stop at the bottom of every page for changing paper. The -n and -o flags may also be quite useful, to start page numbering at a selected point, or to produce only specific pages.

Roffbib understands four command-line number registers, which are something like the two-letter number registers in -ms. The -rN1 argument will number references beginning at one (1); use another number to start somewhere besides one. The -rV2 flag will double-space the entire bibliography, while -rV1 will double-space the references, but single-space the annotation paragraphs. Finally, specifying -rL6i changes the line length from 6.5 inches to 6 inches, and saying -rO1i sets the page offset to one inch, instead of zero. (That's a capital O after -r, not a zero.)

Citing Papers with Refer

The **refer** program normally copies input to output, except when it encounters an item of the form:

```
.[
  partial citation
.]
```

The partial citation may be just an author's name and a date, or perhaps a title and a keyword, or maybe just a document number. **Refer** looks up the citation in the bibliographic database, and transforms it into a full, properly formatted reference. If the partial citation does not correctly identify a single work (either finding nothing, or more than one reference), a diagnostic message is given. If nothing is found, it will say No such paper. If more than one reference is found, it will say Too many hits. Other diagnostic messages can be quite cryptic; if you are in doubt, use **checknr** to verify that all your .'s have matching .'s.

When everything goes well, the reference will be brought in from the database, numbered, and placed at the bottom of the page. This citation, lesk inverted indexes for example, was produced by:

```

This citation,
.[
  lesk inverted indexes
.]
for example, was produced by

```

The `.[` and `.]` markers, in essence, replace the `.FS` and `.FE` of the `-ms` macros, and also provide a numbering mechanism. Footnote numbers will be bracketed on the the line-printer, but superscripted on daisy-wheel terminals and in **troff**. In the reference itself, articles will be quoted, and books and journals will be underlined in **nroff**, and italicized in **troff**.

Sometimes you need to cite a specific page number along with more general bibliographic material. You may have, for instance, a single document that you refer to several times, each time giving a different page citation. This is how you could get p. 10 in the reference:

```

.[
  kies document formatting
  %P 10
.]

```

The first line, a partial citation, will find the reference in your bibliography. The second line will insert the page number into the final citation. Ranges of pages may be specified as `%P 56-78`.

When the time comes to run off a paper, you will need to have two files: the bibliographic database, and the paper to format. Use a command line something like one of these:

```

% refer -p database paper | nroff -ms
% refer -p database paper | tbl | nroff -ms
% refer -p database paper | tbl | neqn | nroff -ms

```

If other preprocessors are used, **refer** should precede **tbl**, which must in turn precede **eqn** or **neqn**. The `-p` option specifies a private database, which most bibliographies are.

Refer's Command-line Options

Many people like to place references at the end of a chapter, rather than at the bottom of the page. The `-e` option will accumulate references until a macro sequence of the form

```

.[
  $LIST$
.]

```

is encountered (or until the end of file). **Refer** will then write out all references collected up to that point, collapsing identical references. Warning: there is a limit (currently 200) on the number of references that can be accumulated at one time.

It is also possible to sort references that appear at the end of text. The `-sKEYS` flag will sort references by fields whose key-letters are in the *KEYS* string, and permute reference numbers in the text accordingly. It is unnecessary to use `-e` with it, since `-s` implies `-e`. Key-letters in *KEYS* may be followed by a '+' to indicate that all such

fields are to be used. The default is to sort by senior author and date, but `-sA+D` will sort on all authors and then date, and `-sA+T` will sort by authors and then title.

Refer can also make citations in what is known as the Social or Natural Sciences format. Instead of numbering references, the `-l` (letter ell) flag makes labels from the senior author's last name and the year of publication. For example, a reference to the paper on Inverted Indexes cited above might appear as [Lesk1978a]. It is possible to control the number of characters in the last name, and the number of digits in the date. For instance, the command line argument `-l6,2` might produce a reference such as [Kernig78c].

Some bibliography standards shun both footnote numbers and labels composed of author and date, requiring some keyword to identify the reference. The `-k` flag indicates that, instead of numbering references, key labels specified on the `%L` line should be used to mark references.

The `-n` flag means to not search the default reference file, located in `/usr/dict/papers/Rv7man`. Using this flag may make **refer** marginally faster. The `-an` flag will reverse the first n author names, printing Jones, J. A. instead of J. A. Jones. Often `-a1` is enough; this will reverse the names of only the senior author. In some versions of **refer** there is also the `-f` flag to set the footnote number to some predetermined value; for example, `-f23` would start numbering with footnote 23.

Making an Index

Once your database is large and relatively stable, it is a good idea to make an index to it, so that references can be found quickly and efficiently. The **indxbib** program makes an inverted index to the bibliographic database (this program is called **pubindex** in the Bell Labs manual). An inverted index could be compared to the thumb cuts of a dictionary instead of going all the way through your bibliography, programs can move to the exact location where a citation is found.

Indxbib itself takes a while to run, and you will need sufficient disk space to store the indexes. But once it has been run, access time will improve dramatically. Furthermore, large databases of several million characters can be indexed with no problem. The program is exceedingly simple to use:

```
% indxbib database
```

Be aware that changing your database will require that you run **indxbib** over again. If you don't, you may fail to find a reference that really is in the database.

Once you have built an inverted index, you can use **lookbib** to find references in the database. **Lookbib** cannot be used until you have run **indxbib**. When editing a paper, **lookbib** is very useful to make sure that a citation can be found as specified. It takes one argument, the name of the bibliography, and then reads partial citations from the terminal, returning references that match, or nothing if none match. Its prompt is the greater-than sign.

```

% lookbib database
> lesk inverted indexes
%A Mike E. Lesk
%T Some Applications of Inverted Indexes on the Unix System
%J Unix Programmer's Manual
%I Bell Laboratories
%C Murray Hill, NJ
%D 1978
%V 2a
%X Difficult to read paper that dwells on indexing strategies,
giving little practical advice about using \fRefer\fP.
>

```

If more than one reference comes back, you will have to give a more precise citation for **refer**. Experiment until you find something that works; remember that it is harmless to overspecify. To get out of the **lookbib** program, type a control-D alone on a line; **lookbib** then exits with an "EOT" message.

Lookbib can also be used to extract groups of related citations. For example, to find all the papers by Brian Kernighan found in the system database, and send the output to a file, type:

```

% lookbib /usr/dict/papers/Ind > kern.refs
> kernighan
> EOT
% cat kern.refs

```

Your file, kern.refs, will be full of references. A similar procedure can be used to pull out all papers of some date, all papers from a given journal, all papers containing a certain group of keywords, etc.

Refer Bugs and Some Solutions

The **refer** program will mess up if there are blanks at the end of lines, especially the %A author line. **Addbib** carefully removes trailing blanks, but they may creep in again during editing. Use an editor command `g/ *$s///` to remove trailing blanks from your bibliography.

Having bibliographic fields passed through as string definitions implies that interpolated strings (such as accent marks) must have two backslashes, so they can pass through copy mode intact. For instance, the word *téléphone* would have to be represented:

```
te\\*'le\\*' phone
```

in order to come out correctly. In the %X field, by contrast, you will have to use single backslashes instead. This is because the %X field is not passed through as a string, but as the body of a paragraph macro.

Another problem arises from authors with foreign names. When a name like Valéry Giscard d'Estaing is turned around by the `-a` option of **refer**, it will appear as d'Estaing, Valéry Giscard, rather than as Giscard d'Estaing, Valéry. To prevent this, enter names as follows:

```
%A Vale\|*`ry Giscard\|Od`Estaing
%A Alexander Csoma\|Ode\|OKo\|*:ro\|*:s
```

(The second is the name of a famous Hungarian linguist.) The backslash-zero is an **nroff/troff** request meaning to insert a digit-width space. It will protect against faulty name reversal, and also against mis-sorting.

Footnote numbers are placed at the end of the line before the `.[` macro. This line should be a line of text, not a macro. As an example, if the line before the `.[` is a `.R` macro, then the `.R` will eat the footnote number. (The `.R` is an `-ms` request meaning change to Roman font.) In cases where the font needs changing, it is necessary to do the following:

```
\fi et al.\fR
.[
awk aho kernighan weinberger
.]
```

Now the reference will be to Aho *et al.* awk aho kernighan. The `\fi` changes to italics, and the `\fR` changes back to Roman font. Both these requests are **nroff/troff** requests, not part of `-ms`. If and when a footnote number is added after this sequence, it will indeed appear in the output.

Internal Details of Refer

You have already read everything you need to know in order to use the **refer** bibliography system. The remaining sections are provided only for extra information, and in case you need to change the way **refer** works.

The output of **refer** is a stream of string definitions, one for each field in a reference. To create string names, percent signs are simply changed to an open bracket, and an `[F` string is added, containing the footnote number. The `%X`, `%Y` and `%Z` fields are ignored; however, the **annobib** program changes the `%X` to an `.AP` (annotation paragraph) macro. The citation used above yields this intermediate output:

```
.ds [F 1
.-
.ds [A Mike E. Lesk
.ds [T Some Applications of Inverted Indexes on the Unix System
.ds [J Unix Programmer's Manual
.ds [I Bell Laboratories
.ds [C Murray Hill, NJ
.ds [D 1978
.ds [V 2a
.nr [T 0
.nr [A 0
.nr [O 0
.][ 1 journal-article
```

These string definitions are sent to **nroff**, which can use the `-ms` macros defined in `/usr/lib/mx/tmac.xref` to take care of formatting things properly. The initializing macro `.[-` precedes the string definitions, and the labeled macro `.][` follows. These are changed from the input `.[` and `.]` so that running a file twice through **refer** is harmless.

The `.|` macro, used to print the reference, is given a type-number argument, which is a numeric label indicating the type of reference involved. Here is a list of the various kinds of references:

Field	Value	Kind of Reference
<code>%J</code>	1	Journal Article
<code>%B</code>	3	Article in Book
<code>%R %G</code>	4	Report, Government Report
<code>%I</code>	2	Book
<code>%M</code>	5	Bell Labs Memorandum (undefined)
none	0	Other

The order listed above is indicative of the precedence of the various fields. In other words, a reference that has both the `%J` and `%B` fields will be classified as a journal article. If none of the fields listed is present, then the reference will be classified as other.

The footnote number is flagged in the text with the following sequence, where *number* is the footnote number:

```
\*([.number\*(.)
```

The `*([.` and `*(.)` stand for bracketing or superscripting. In **nroff** with low-resolution devices such as the `lpr` and a crt, footnote numbers will be bracketed. In **troff**, or on daisy-wheel printers, footnote numbers will be superscripted. Punctuation normally comes before the reference number; this can be changed by using the `-P` (postpunctuation) option of **refer**.

In some cases, it is necessary to override certain fields in a reference. For instance, each time a work is cited, you may want to specify different page numbers, and you may want to change certain fields. This citation will find the Lesk reference, but will add specific page numbers to the output, even though no page numbers appeared in the original reference.

```
.|
lesk inverted indexes
%P 7-13
%I Computing Services
%O UNX 12.2.2.
.]
```

The `%I` line will also override any previous publisher information, and the `%O` line will append some commentary. The **refer** program simply adds the new `%P`, `%I`, and `%O` strings to the output, and later strings definitions cancel earlier ones.

It is also possible to insert an entire citation that does not appear in the bibliographic database. This reference, for example, could be added as follows:

```
.]
%A    Brian Kernighan
%T    A Troff Tutorial
%l    Bell Laboratories
%D    1978
.]
```

This will cause **refer** to interpret the fields exactly as given, without searching the bibliographic database. This practice is not recommended, however, because it's better to add new references to the database, so they can be used again later.

If you want to change the way footnote numbers are printed, signals can be given on the `.[` and `.]` lines. For example, to say See reference (2), the citation should appear as:

```
See reference
.[(
partial citation
.]),
```

Note that blanks are significant on these signal lines. If a permanent change in the footnote format is desired, it's best to redefine the `[.` and `.]` strings.

Changing the Refer Macros

This section is provided for those who wish to rewrite or modify the **refer** macros. This is necessary in order to make output correspond to specific journal requirements, or departmental standards. First there is an explanation of how new macros can be substituted for the old ones. Then several alterations are given as examples. Finally, there is an annotated copy of the **refer** macros used by **roffbib**.

The **refer** macros for **nroff/troff** supplied by the `-ms` macro package reside in `/usr/lib/mx/tmac.xref`; they are reference macros, for producing footnotes or endnotes. The **refer** macros used by **roffbib**, on the other hand, reside in `/usr/lib/tmac/tmac.bib`; they are for producing a stand-alone bibliography.

To change the macros used by **roffbib**, you will need to get your own version of this shell script into the directory where you are working. These two commands will get you a copy of **roffbib** and the macros it uses: †

```
% cp /usr/lib/tmac/tmac.bib bibmac
```

You can proceed to change `bibmac` as much as you like. Then when you use **roffbib**, you should specify your own version of the macros, which will be substituted for the normal ones

```
% roffbib -m bibmac filename
```

where *filename* is the name of your bibliography file. Make sure there's a space between `-m` and **bibmac**.

If you want to modify the **refer** macros for use with **nroff** and the `-ms` macros, you will need to get a copy of `tmac.xref`:

```
% cp /usr/lib/ms/s.ref refmac
```

These macros are much like `bibmac`, except they have `.FS` and `.FE` requests, to be used

in conjunction with the `-ms` macros, rather than independently defined `.XP` and `.AP` requests. Now you can put this line at the top of the paper to be formatted:

```
.so refmac
```

Your new **refer** macros will override the definitions previously read in by the `-ms` package. This method works only if `refmac` is in the working directory.

Suppose you didn't like the way dates are printed, and wanted them to be parenthesized, with no comma before. There are five identical lines you will have to change. The first line below is the old way, while the second is the new way:

```
.if !"\\*(D"" , \\*(D\c
.if !"\\*(D"" \& (\\*(D)\c
```

In the first line, there is a comma and a space, but no parentheses. The `\c` at the end of each line indicates to **nroff** that it should continue, leaving no extra space in the output. The `\&` in the second line is the do-nothing character; when followed by a space, a space is sent to the output.

If you need to format a reference in the style favored by the Modern Language Association or Chicago University Press, in the form (city: publisher, date), then you will have to change the middle of the book macro [2 as follows:

```
\& (\c
.if !"\\*(C"" \\*(C:
\\*(I\c
.if !"\\*(D"" , \\*(D\c
)\c
```

This would print (Berkeley: Computing Services, 1982) if all three strings were present. The first line prints a space and a parenthesis; the second prints the city (and a colon) if present; the third always prints the publisher (books must have a publisher, or else they're classified as other); the fourth line prints a comma and the date if present; and the fifth line closes the parentheses. You would need to make similar changes to the other macros as well.

C

C

C

C

C

READER'S RESPONSE

We use readers' comments in revising and improving our documents.

Document Title: DOMAIN/IX Text Processing Guide
Order Number: 005802
Revision: 00
Date of Publication: July, 1985

What is the best feature of this manual?

Three horizontal lines for writing the best feature of the manual.

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible.)

Four horizontal lines for listing errors, omissions, or problem areas.

What type of user are you?

- List of user types with checkboxes: Systems programmer; language, Applications programmer; language, Manager/Professional, Technical Professional, Administrative/Support Personnel, Student programmer, User with little programming experience, Other.

How often do you use your system?

One horizontal line for frequency of system use.

Nature of your work on the DOMAIN System:

One horizontal line for nature of work.

Your name

Date

Organization

Street Address

City

State

Zip/Country

No postage necessary if mailed in the U.S. Fold on dotted lines (see reverse), tape, and mail.

cut or fold along dotted line

FOLD



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE



APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

FOLD

READER'S RESPONSE

We use readers' comments in revising and improving our documents.

Document Title: DOMAIN/IX Text Processing Guide
Order Number: 005802
Revision: 00
Date of Publication: July, 1985

What is the best feature of this manual?

Three horizontal lines for writing the best feature of the manual.

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible.)

Three horizontal lines for listing errors, omissions, or problem areas.

What type of user are you?

- Systems programmer; language
Applications programmer; language
Manager/Professional
Technical Professional
Administrative/Support Personnel
Student programmer
User with little programming experience
Other

How often do you use your system?

One horizontal line for frequency of system use.

Nature of your work on the DOMAIN System:

One horizontal line for nature of work.

Your name Date

Organization

Street Address

City State Zip/Country

No postage necessary if mailed in the U.S. Fold on dotted lines (see reverse), tape, and mail.

cut or fold along dotted line

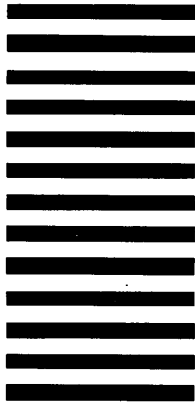
FOLD



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE



APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

FOLD

Instruction Sheet

Insert Tabbed Divider Page:

Editors
Formatters
Appendices

Before Page:

Chapter 1: An ed Tutorial
Chapter 1: A troff Tutorial
Appendix A: Advanced Editing on UNIX

C

C

C

C

C