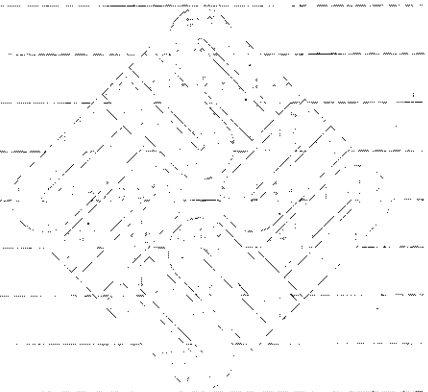
sun microsystems

Editing and Text Processing *on the* Sun Workstation

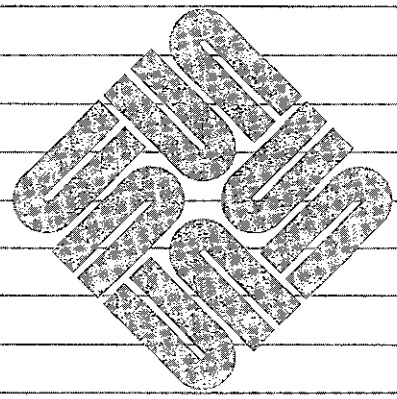


Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Part No: 800-1787-07
Revision A of 15 [illegible] 1988

sun microsystems

Editing and Text Processing *on the* Sun Workstation



Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Part No: 800-1174-01
Revision D of 15 May, 1985

Credits and Acknowledgements

Material in this *Editing and Text Processing on the Sun Workstation* comes from a number of sources: *An Introduction to Display Editing with Vi*, William Joy, University of California, Berkeley, revised by Mark Horton; *Vi Command and Function Reference*, Alan P. W. Hewett, revised by Mark Horton; *Ex Reference Manual*, William Joy, revised by Mark Horton, University of California, Berkeley; *Awk — A Pattern Scanning and Processing Language*, Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger, Bell Laboratories, Murray Hill, New Jersey; *Edit: A Tutorial*, Ricki Blau, James Joyce, University of California, Berkeley; *A Tutorial Introduction to the UNIX Text Editor*, Brian W. Kernighan, Bell Laboratories, Murray Hill, New Jersey; *Advanced Editing on UNIX*, Brian W. Kernighan, Bell Laboratories, Murray Hill, New Jersey; *Sed — a Non-Interactive Text Editor*, Lee. E. McMahon, Bell Laboratories, Murray Hill, New Jersey; *Nroff/Troff User's Manual*, Joseph F. Ossanna, Bell Laboratories, Murray Hill, New Jersey; *A Troff Tutorial*, Brian W. Kernighan, Bell Laboratories, Murray Hill, New Jersey; *Typing Documents on the UNIX System: Using the -ms Macros with Troff and Nroff*, M. E. Lesk, Bell Laboratories, Murray Hill, New Jersey; *A Guide to Preparing Documents with -ms*, M. E. Lesk, Bell Laboratories, Murray Hill, New Jersey; *Document Formatting on UNIX Using the −ms Macros*, Joel Kies, University of California, Berkeley, California; *Tbl — A Program to Format Tables*, M. E. Lesk, Bell Laboratories, Murray Hill, New Jersey; *A System for Typesetting Mathematics*, Brian W. Kernighan, Lorinda L. Cherry, Bell Laboratories, Murray Hill, New Jersey; *Typesetting Mathematics — User's Guide*, Brian W. Kernighan, Lorinda L. Cherry, Bell Laboratories, Murray Hill, New Jersey; *Writing Tools — The Style and Diction Programs*, L. L. Cherry, W. Vesterman, Bell Laboratories, Murray Hill, New Jersey; *Updating Publications Lists*, M. E. Lesk, Bell Laboratories, Murray Hill, New Jersey; *Some Applications of Inverted Indexes on the UNIX System*, M. E. Lesk, Bell Laboratories, Murray Hill, New Jersey; *Writing Papers with Nroff Using -me*, Eric P. Allman, University of California, Berkeley; and *-me Reference Manual*, Eric P. Allman, University of California, Berkeley. *Introducing the UNIX System*, Henry McGilton, Rachel Morgan, McGraw-Hill Book Company, 1983. These materials are gratefully acknowledged.

Trademarks

Revision History

Version	Date	Comments
A	15 May 1983	First release of Editing and Text Processing.
B	1 November 1983	Updated and reorganized.
C	7 January 1984	New **−ms** macros; additions to document preparation introduction; minor corrections.
D	15 May 1985	2.0 FCS release (old format)

Contents

Chapter 1 An Introduction to Text Editing **1-1**

Chapter 2 Using `vi`, the Visual Display Editor **2-1**

Chapter 3 Command Reference for the **ex** Line Editor **3-1**

Chapter 4 Using the **ed** Line Editor **4-1**

Chapter 5 Using **sed**, the Stream Text Editor **5-1**

Chapter 6 Pattern Scanning and Processing with **awk** **6-1**

Chapter 7 Introduction to Document Preparation **7-1**

Chapter 8 Formatting Documents with the **—ms** Macros **8-1**

Chapter 9 The **—man** Macro Package **9-1**

Chapter 10 Formatting Tables with tbl **10-1**

Chapter 11 PIC — A Graphics Language for Typesetting **11-1**

Chapter 12 Typesetting Mathematics with eqn **12-1**

Chapter 13 Refer — A Bibliography System **13-1**

Chapter 14 Formatting Documents with the —me Macros **14-1**

Chapter 15 Formatting Documents with ***nroff*** and ***troff*** **15-1**

Appendix A Examples of Fonts and Non—ASCII Characters **A-1**

Appendix B ***troff*** Request Summary **B-1**

Appendix C Escape Sequences for Characters, Indicators, and Functions **C-1**

Appendix D Predefined Number Registers **D-1**

Appendix E Description of ***troff*** Output Codes **E-1**

Contents

Preface xxvii

Chapter 1 An Introduction to Text Editing **1-1**
1.1. Sun System Editors 1-1
1.2. Text Editing Basics 1-2
1.2.1. Regular Expressions in Text Patterns 1-3
1.3. What to Do If Something Goes Wrong 1-5

Chapter 2 Using `vi`, the Visual Display Editor **2-1**
2.1. `vi` and `ex` 2-1
2.2. Getting Started 2-2
2.2.1. Editing a File 2-2
2.2.2. The Editor's Copy — Editing in the Buffer 2-2
2.2.3. Arrow Keys 2-3
2.2.4. Special Characters: ESC, CR and CTRL-C 2-3
2.2.5. Getting Out of `vi` — `:q`, `:q!`, `:w`, `ZZ`, `:wq` 2-3
2.3. Moving Around in the File 2-4
2.3.1. Scrolling and Paging — CTRL-D, CTRL-U, CTRL-E, CTRL-Y, CTRL-F, CTRL-B 2-4
2.3.2. Searching, Goto, and Previous Context — `/`, `?`, `G` 2-4
2.3.3. Moving Around on the Screen — `h`, `j`, `k`, `l` 2-6
2.3.4. Moving Within a Line — `b`, `w`, `e`, `E`, `B`, `W` 2-6
2.3.5. Viewing a File — `view` 2-6
2.4. Making Simple Changes 2-7
2.4.1. Inserting — `i` and `a` 2-7
2.4.2. Making Small Corrections — `x`, `r`, `s`, `R` 2-8
2.4.3. Deleting, Repeating, and Changing — `dw`, `.`, `db`, `c` 2-8
2.4.4. Operating on Lines — `dd`, `cc`, `S` 2-9
2.4.5. Undoing — `u`, `U` 2-9
2.5. Moving About: Rearranging and Duplicating Text 2-10
2.5.1. Low-level Character Motions — `f`, `F`, `^` 2-10
2.5.2. Higher Level Text Objects — `(`, `)`, `{`, `}`, `[[`, `]]` 2-10
2.5.3. Rearranging and Duplicating Text — `y`, `Y`, `p`, `P` 2-11
2.6. High-Level Commands 2-12
2.6.1. Writing, Quitting, and Editing New Files — `ZZ`, `:w`, `:q`, `:e`, `:n` 2-12
2.6.2. Escaping to a Shell — `:!`, `:sh`, CTRL-Z 2-12

2.6.3. Marking and Returning — m 2-13
2.6.4. Adjusting the Screen CTRL-L, z 2-13
2.7. Special Topics 2-13
2.7.1. Options, the Set Variable, and Editor Start-up Files 2-13
2.7.2. Recovering Lost Lines 2-15
2.7.3. Recovering Lost Files — the -r Option 2-15
2.7.4. Continuous Text Input — wrapmargin 2-16
2.7.5. Features for Editing Programs 2-16
2.7.6. Filtering Portions of the Buffer 2-17
2.7.7. Commands for Editing LISP 2-17
2.7.8. Macros 2-17
2.7.9. Word Abbreviations — :ab, :una 2-19
2.8. Nitty-gritty Details 2-19
2.8.1. Line Representation in the Display 2-19
2.8.2. Command Counts 2-20
2.8.3. File Manipulation Commands 2-20
2.8.4. More about Searching for Strings 2-22
2.8.5. More about Input Mode 2-23
2.9. Command and Function Reference 2-24
2.9.1. Notation 2-24
2.9.2. Commands 2-25
2.9.3. Entry and Exit 2-25
2.9.4. Cursor and Page Motion 2-25
2.9.5. Searches 2-28
2.9.6. Text Insertion 2-28
2.9.7. Text Deletion 2-29
2.9.8. Text Replacement 2-29
2.9.9. Moving Text 2-30
2.9.10. Miscellaneous Commands 2-31
2.9.11. Special Insert Characters 2-32
2.9.12. : Commands 2-32
2.9.13. Set Commands 2-33
2.9.14. Character Functions 2-37
2.10. Terminal Information 2-44
2.10.1. Specifying Terminal Type 2-45
2.10.2. Special Arrangements for Startup 2-46
2.10.3. Open Mode on Hardcopy Terminals and 'Glass tty's' 2-46
2.10.4. Editing on Slow Terminals 2-47
2.10.5. Upper-case Only Terminals 2-48
2.11. Command Summary 2-48

Chapter 3 Command Reference for the **ex** Line Editor **3-1**
3.1. Using **ex** 3-1
3.2. File Manipulation 3-2
3.2.1. Current File 3-2
3.2.2. Alternate File 3-2

3.2.3. Filename Expansion 3-2
3.3. Special Characters 3-3
3.3.1. Multiple Files and Named Buffers 3-3
3.3.2. Read Only Mode 3-3
3.4. Exceptional Conditions 3-3
3.4.1. Errors and Interrupts 3-3
3.4.2. Recovering If Something Goes Wrong 3-4
3.5. Editing Modes 3-4
3.6. Command Structure 3-4
3.6.1. Specifying Command Parameters 3-4
3.6.2. Invoking Command Variants 3-5
3.6.3. Flags after Commands 3-5
3.6.4. Writing Comments 3-5
3.6.5. Putting Multiple Commands on a Line 3-5
3.6.6. Reporting Large Changes 3-5
3.7. Command Addressing 3-6
3.7.1. Addressing Primitives 3-6
3.7.2. Combining Addressing Primitives 3-6
3.8. Regular Expressions and Substitute Replacement Patterns 3-6
3.8.1. Regular Expressions 3-7
3.8.2. Magic and Nomagic 3-7
3.8.3. Basic Regular Expression Summary 3-7
3.8.4. Combining Regular Expression Primitives 3-8
3.8.5. Substitute Replacement Patterns 3-8
3.9. Command Reference 3-8
3.10. Option Descriptions 3-17
3.11. Limitations 3-22

Chapter 4 Using the **ed** Line Editor **4-1**
4.1. Getting Started 4-1
4.1.1. Creating Text — the Append Command **a** 4-2
4.1.2. Error Messages — **?** 4-2
4.1.3. Writing Text Out as a File — the Write Command **w** 4-3
4.1.4. Leaving **ed** — the Quit Command **q** 4-4
4.1.5. Creating a New File — the Edit Command **e** 4-4
4.1.6. Exercise: Trying the **e** Command 4-5
4.1.7. Checking the Filename — the Filename Command **f** 4-6
4.1.8. Reading Text from a File — the Read Command **r** 4-6
4.1.9. Printing the Buffer Contents — the Print Command **p** 4-7
4.1.10. Exercise: Trying the **p** Command 4-8
4.1.11. Displaying Text — the List Command **l** 4-8
4.1.12. The Current Line — 'Dot' or '.' 4-9
4.1.13. Deleting Lines — the Delete Command **d** 4-10
4.1.14. Exercise: Experimenting 4-10
4.1.15. Modifying Text — the Substitute Command **s** 4-11
4.1.16. The Ampersand **&** 4-13

4.1.17. Exercise: Trying the `s` and `g` Commands 4-14
4.1.18. Undoing a Command — the Undo Command `u` 4-14
4.2. Changing and Inserting Text — the `c` and `i` Commands 4-14
4.2.1. Exercise: Trying the `c` Command 4-15
4.3. Specifying Lines in the Editor 4-16
4.3.1. Context Searching 4-16
4.3.2. Exercise: Trying Context Searching 4-17
4.3.3. Specifying Lines with Address Arithmetic — `+` and `-` 4-17
4.3.4. Repeated Searches — `//` and `??` 4-18
4.3.5. Default Line Numbers and the Value of Dot 4-19
4.3.6. Combining Commands — the Semicolon `;` 4-21
4.3.7. Interrupting the Editor 4-22
4.4. Editing All Lines — the Global Commands `g` and `v` 4-22
4.4.1. Multi-line Global Commands 4-23
4.5. Special Characters 4-24
4.5.1. Matching Anything — the Dot '.' 4-24
4.5.2. Specifying Any Character — the Backslash '\' 4-25
4.5.3. Specifying the End of Line — the Dollar Sign `$` 4-27
4.5.4. Specifying the Beginning of the Line — the Circumflex `^` 4-28
4.5.5. Matching Anything — the Star `*` 4-28
4.5.6. Character Classes — Brackets `[ ]` 4-30
4.6. Cutting and Pasting with the Editor 4-31
4.6.1. Moving Lines Around 4-31
4.6.2. Moving Text Around — the Move Command `m` 4-31
4.6.3. Substituting Newlines 4-33
4.6.4. Joining Lines — the Join Command `j` 4-33
4.6.5. Rearranging a Line with `\( ... \)` 4-34
4.6.6. Marking a Line — the Mark Command `k` 4-34
4.6.7. Copying Lines — the Transfer Command `t` 4-35
4.7. Escaping to the Shell with `!` 4-35
4.8. Supporting Tools 4-35
4.8.1. Editing Scripts 4-36
4.8.2. Matching Patterns with `grep` 4-36
4.9. Summary of Commands and Line Numbers 4-37

Chapter 5 Using `sed`, the Stream Text Editor **5-1**
5.1. Using `sed` 5-2
5.1.1. Command Options 5-2
5.2. Editing Commands Application Order 5-3
5.3. Specifying Lines for Editing 5-3
5.3.1. Line-number Addresses 5-4
5.3.2. Context Addresses 5-4
5.3.3. Number of Addresses 5-5
5.4. Functions 5-6
5.4.1. Whole Line Oriented Functions 5-6
5.4.2. The Substitute Function `s` 5-7

5.4.3. Input-output Functions 5-9
5.4.4. Multiple Input-line Functions 5-10
5.4.5. Hold and Get Functions 5-10
5.4.6. Flow-of-Control Functions 5-11
5.4.7. Miscellaneous Functions 5-12

Chapter 6 Pattern Scanning and Processing with `awk` **6-1**
6.1. Using `awk` 6-2
6.1.1. Program Structure 6-2
6.1.2. Records and Fields 6-3
6.2. Displaying Text 6-3
6.3. Specifying Patterns 6-5
6.3.1. `BEGIN` and `END` 6-5
6.3.2. Regular Expressions 6-5
6.3.3. Relational Expressions 6-6
6.3.4. Combinations of Patterns 6-7
6.3.5. Pattern Ranges 6-7
6.4. Actions 6-7
6.4.1. Assignments, Variables, and Expressions 6-7
6.4.2. Field Variables 6-8
6.4.3. String Concatenation 6-9
6.4.4. Built-in Functions 6-9
6.4.4.1. `length` Function 6-9
6.4.4.2. `substring` Function 6-10
6.4.4.3. `index` Function 6-10
6.4.4.4. `sprintf` Function 6-10
6.4.5. Arrays 6-10
6.4.6. Flow-of-Control Statements 6-11

Chapter 7 Introduction to Document Preparation **7-1**
7.1. What Do Text Formatters Do? 7-1
7.2. What is a Macro Package? 7-2
7.3. What is a Preprocessor? 7-2
7.4. Typesetting Jargon 7-3
7.5. Hints for Typing in Text 7-4
7.6. Types of Paragraphs 7-4
7.7. Quick References 7-8
7.7.1. Displaying and Printing Documents 7-8
7.7.2. Technical Memorandum 7-9
7.7.3. Section Headings for Documents 7-11
7.7.4. Changing Fonts 7-11
7.7.5. Making a Simple List 7-11
7.7.6. Multiple Indents for Lists and Outlines 7-12
7.7.7. Displays 7-13
7.7.8. Footnotes 7-13
7.7.9. Keeping Text Together — Keeps 7-13
7.7.10. Double-Column Format 7-14
7.7.11. Sample Tables 7-14
7.7.12. Writing Mathematical Equations 7-16
7.7.13. Registers You Can Change 7-17

Chapter 8 Formatting Documents with the **—ms** Macros **8-1**
8.1. Changes in the New **—ms** Macro Package 8-1
8.2. Displaying and Printing Documents with —ms 8-1
8.3. What Can Macros Do? 8-2
8.4. Formatting Requests 8-2
8.4.1. Paragraphs 8-3
8.4.1.1. Standard Paragraph — .PP 8-3
8.4.1.2. Left-Block Paragraph — .LP 8-3
8.4.1.3. Indented Paragraph — .IP 8-3
8.4.1.4. Nested Indentation — .RS and .RE 8-5
8.4.1.5. Quoted Paragraph — .QP 8-6
8.4.2. Section Headings — .SH and .NH 8-6
8.4.3. Cover Sheets and Title Pages 8-7
8.4.4. Running Heads and Feet — LH, CH, RH 8-8
8.4.5. Custom Headers and Footers — .OH, .EH, .OF, and .EF 8-9

8.4.6. Multi-column Formats — .2C and .MC 8-9
8.4.7. Footnotes — .FS and .FE 8-11
8.4.8. Endnotes 8-12
8.4.9. Displays and Tables — .DS and .DE 8-12
8.4.10. Keeping Text Together — .KS, .KE and .KF 8-13
8.4.11. Boxing Words or Lines — .BX and .B1 and .B2 8-13
8.4.12. Changing Fonts — .I, .B, .R and .UL 8-14
8.4.13. Changing the Type Size — .LG, .SM and .NL 8-14
8.4.14. Dates — .DA and .ND 8-15
8.4.15. Thesis Format Mode — .TM 8-15
8.4.16. Bibliography — .XP 8-15
8.4.17. Table of Contents — .XS, .XE, .XA, .PX 8-16
8.4.18. Defining Quotation Marks 8-16
8.4.19. Accent Marks 8-16
8.5. Modifying Default Features 8-18
8.5.1. Dimensions 8-18
8.6. Using nroff and troff Requests 8-20
8.7. Using **–ms** with tbl to Format Tables 8-21
8.8. Using **–ms** with eqn to Typeset Mathematics 8-21
8.9. Register Names 8-22
8.10. Order of Requests in Input 8-23
8.11. **–ms** Request Summary 8-24

Chapter 9 The **–man** Macro Package **9-1**
9.1. Parts of a Manual Page 9-1
9.2. Coding Conventions 9-2
9.2.1. The .TH Line — Identifying the Page 9-2
9.2.2. The **NAME** Line 9-2
9.2.3. The **SYNOPSIS** Section 9-3
9.2.4. The **DESCRIPTION** Section 9-3
9.2.5. The **OPTIONS** Section 9-4
9.2.6. The **FILES** Section 9-6
9.2.7. The **SEE ALSO** Section 9-6
9.2.8. The **BUGS** Section 9-7
9.3. New Features of the **–man** Macro Package 9-7
9.3.1. New Number Registers 9-7
9.3.2. Using the Number Registers 9-8
9.4. How to Format a Manual Page 9-8
9.5. Summary of the **–man** Macro Package Requests 9-9

Chapter 10 Formatting Tables with tbl **10-1**
10.1. Running tbl 10-2
10.2. Input Commands 10-4
10.2.1. Options that Affect the Whole Table 10-4
10.2.2. Key Letters — Format Describing Data Items 10-5
10.2.3. Optional Features of Key Letters 10-6

10.2.4. Data to be Formatted in the Table 10-8
10.2.5. Changing the Format of a Table 10-9
10.3. Examples 10-10
10.4. Tbl Commands 10-21

Chapter 11 PIC — A Graphics Language for Typesetting **11-1**
11.1. Introduction 11-1
11.2. Basics 11-1
11.3. Controlling Sizes of Objects 11-5
11.3.1. Variables for Controlling Size of Objects 11-8
11.4. Controlling Positions of Objects 11-9
11.5. Labels and Corners 11-10
11.6. Variables and Expressions 11-14
11.7. More on Text 11-15
11.8. Lines and Splines 11-15
11.9. Blocks 11-16
11.10. Macros 11-18
11.11. TROFF Interface 11-19
11.12. Some Examples 11-20
11.13. Final Observations 11-23
11.13.1. Acknowledgements 11-24
11.14. PIC Reference Manual 11-25
11.14.1. Pictures 11-25
11.14.2. Elements 11-25
11.14.3. Primitives 11-26
11.14.4. Attributes 11-26
11.14.5. Text 11-27
11.14.6. Positions and places 11-27
11.14.7. Variables 11-28
11.14.8. Expressions 11-29
11.14.9. Definitions 11-29

Chapter 12 Typesetting Mathematics with eqn **12-1**
12.1. Displaying Equations — '.EQ' and '.EN' 12-1
12.2. Running eqn and neqn 12-2
12.3. Putting Spaces in the Input Text 12-3
12.4. Producing Spaces in the Output Text 12-4
12.5. Symbols, Special Names, and Greek Letters 12-5
12.6. Subscripts and Superscripts — 'sub' and 'sup' 12-5
12.7. Grouping Equation Parts — '{' and '}' 12-6
12.8. Fractions — 'over' 12-7
12.9. Square Roots — 'sqrt' 12-8
12.10. Summation, Integral, and Other Large Operators 12-9
12.11. Size and Font Changes 12-9
12.12. Diacritical Marks 12-11
12.13. Quoted Text 12-11

12.14. Lining Up Equations — 'mark' and 'lineup' 12-12
12.15. Big Brackets 12-13
12.16. Piles — 'pile' 12-13
12.17. Matrices — 'matrix' 12-14
12.18. Shorthand for In-line Equations — 'delim' 12-15
12.19. Definitions — 'define' 12-15
12.20. Tuning the Spacing 12-17
12.21. Troubleshooting 12-17
12.22. Precedences and Keywords 12-18
12.23. Several Examples 12-22

Chapter 13 Refer — A Bibliography System **13-1**
13.1. Introduction 13-1
13.2. Features 13-1
13.3. Data Entry with Addbib 13-3
13.4. Printing the Bibliography 13-4
13.5. Citing Papers with Refer 13-4
13.6. Refer's Command-line Options 13-6
13.7. Making an Index 13-6
13.8. Refer Bugs and Some Solutions 13-7
13.8.1. Blanks at Ends of Lines 13-7
13.8.2. Interpolated Strings 13-8
13.8.3. Interpreting Foreign Surnames 13-8
13.8.4. Footnote Numbers 13-8
13.9. Internal Details of Refer 13-9
13.10. Changing the Refer Macros 13-10

Chapter 14 Formatting Documents with the —me Macros **14-1**
14.1. Using —me 14-1
14.2. Basic —me Requests 14-2
14.2.1. Paragraphs 14-2
14.2.1.1. Standard Paragraph — '.pp' 14-2
14.2.1.2. Left Block Paragraphs — '.lp' 14-3
14.2.1.3. Indented Paragraphs — '.ip' and '.np' 14-3
14.2.1.4. Paragraph Reference 14-5
14.3. Headers and Footers — '.he' and '.fo' 14-5
14.3.1. Headers and Footers Reference 14-6
14.3.2. Double Spacing — '.ls 2' 14-6
14.3.3. Page Layout 14-7
14.3.4. Underlining — '.ul' 14-8
14.3.5. Displays 14-8
14.3.5.1. Major Quotes — '.(q' and '.)q' 14-8
14.3.5.2. Lists — '.(l' and '.)l' 14-9
14.3.5.3. Keeps — '.(b' and '.)b', '.(z' and '.)z' 14-9
14.4. Fancy Displays 14-10
14.4.1. Display Reference 14-11

14.4.2. Annotations 14-12
14.4.3. Footnotes — '.(f' and '.)f' 14-12
14.4.4. Delayed Text 14-13
14.4.5. Indexes — '.(x' '.)x' and '.xp' 14-13
14.4.6. Annotations Reference 14-14
14.5. Fancy Features 14-14
14.5.1. Section Headings — '.sh' and '.uh' 14-15
14.5.1.1. Section Heading Reference 14-16
14.5.2. Parts of the Standard Paper 14-17
14.5.2.1. Standard Paper Reference 14-18
14.5.3. Two-Column Output — '.2c' 14-19
14.5.3.1. Columned Output Reference 14-20
14.5.4. Defining Macros — '.de' 14-20
14.5.5. Annotations Inside Keeps 14-20
14.6. Using 'troff' for Phototypsetting 14-21
14.6.1. Fonts 14-21
14.6.2. Point Sizes — '.sz' 14-23
14.6.2.1. Fonts and Sizes Reference 14-23
14.6.3. Quotes — '*(lq' and '*(rq' 14-23
14.7. Adjusting Macro Parameters 14-24
14.8. Roff Support 14-25
14.9. Preprocessor Support 14-25
14.10. Predefined Strings 14-26
14.11. Miscellaneous Requests 14-26
14.12. Special Characters and Diacritical Marks — '.sc' 14-27
14.13. '–me' Request Summary 14-28

Chapter 15 Formatting Documents with ***nroff*** and ***troff*** **15-1**
15.1. Introduction to nroff and troff 15-1
15.1.1. Text Formatting Versus Word Processing 15-2
15.1.2. The Evolution of nroff and troff 15-3
15.1.3. Preprocessors and Postprocessors 15-3
15.1.4. troff, Typesetters, and Special-Purpose Formatters 15-4
15.1.5. Using the nroff and troff Text Formatters 15-4
15.1.5.1. Options Common to nroff and troff 15-5
15.1.5.2. Options Applicable Only to nroff 15-5
15.1.5.3. Options Applicable Only to troff 15-6
15.1.6. General Explanation of troff and nroff Source Files 15-6
15.1.6.1. Backspacing 15-7
15.1.6.2. Comments 15-7
15.1.6.3. Continuation Lines 15-8
15.1.6.4. Transparent Throughput 15-8
15.1.6.5. Formatter and Device Resolution 15-8
15.1.6.6. Specifying Numerical Parameters 15-8
15.1.6.7. Numerical Expressions 15-9
15.1.7. Notation Used in this Manual 15-10

15.1.8. Output and Error Messages 15-11
15.2. Filling and Adjusting Lines of Text 15-12
15.2.1. Controlling Line Breaks 15-13
15.2.1.1. **.br** — Break Lines 15-14
15.2.2. Continuation Lines and Interrupted Text 15-14
15.2.3. **.ad** — Specify Adjusting Styles 15-15
15.2.4. **.na** — No Adjusting 15-16
15.2.5. **.nf** and **.fi** — Turn Filling Off and On 15-17
15.2.6. Hyphenation 15-18
15.2.6.1. **.nh** and **.hy** — Control Hyphenation 15-18
15.2.6.2. **.hw** — Specify Hyphenation Word List 15-19
15.2.6.3. **.hc** — Specify Hyphenation Character 15-20
15.2.7. **.ce** — Center Lines of Text 15-20
15.2.8. **.ul** and **.cu** — Underline or Emphasize Text 15-21
15.2.9. Underlining 15-22
15.3. Controlling Page Layout 15-24
15.3.1. Margins and Indentations 15-27
15.3.1.1. **.po** — Set Page Offset 15-27
15.3.1.2. **.ll** — Set Line Length 15-27
15.3.1.3. **.in** — Set Indent 15-28
15.3.1.4. **.ti** — Temporararily Indent One Line 15-30
15.3.2. Page Lengths, Page Breaks, and Conditional Page Breaks 15-32
15.3.2.1. **.pl** — Set Page Length 15-32
15.3.2.2. **.bp** — Start a New Page 15-33
15.3.2.3. **.pn** — Set Page Number 15-33
15.3.2.4. **.ne** — Specify Space Needed 15-34
15.3.3. Multi-Columnar Page Layout by Marking and Returning 15-34
15.3.3.1. **.mk** — Mark Current Vertical Position 15-35
15.3.3.2. **.rt** — Return to Marked Vertical Position 15-35
15.4. Line Spacing and Character Sizes 15-36
15.4.1. **.sp** — Get Extra Space 15-36
15.4.2. **.ls** — Change Line Spacing 15-37
15.4.3. **\x** Function — Get Extra Line-space 15-37
15.4.4. **.vs** — Change Vertical Distance Between Lines 15-38
15.4.5. **.sp** — Get Blocks of Vertical Space 15-38
15.4.6. **.sv** — Save Block of Vertical Space 15-39
15.4.7. **.os** — Output Saved Vertical Space 15-39
15.4.8. **.ns** — Set No Space Mode 15-40
15.4.9. **.ps** — Change the Size of the Type 15-40
15.4.10. **.ss** — Set Size of Space Character 15-42
15.4.11. **.cs** — Set Constant Width Characters 15-43
15.5. Fonts and Special Characters 15-44
15.5.1. Character Set 15-46
15.5.2. Fonts 15-46
15.5.3. **.bd** — Artificial Bold Face 15-47
15.5.4. **.ft** — Set Font 15-47

15.5.5. **.fp** — Set Font Position 15-48
15.5.6. **.fz** — Force Font Size 15-48
15.5.7. **.lg** — Control Ligatures 15-48
15.6. Tabs, Leaders, and Fields — Aligning Things in Columns 15-50
15.6.1. **.ta** — Set Tabs 15-50
15.6.1.1. Setting Relative Tab Stops 15-50
15.6.1.2. Right-Adjusted Tab Stops 15-51
15.6.1.3. Centered Tab Stops 15-51
15.6.1.4. **.tc** — Change Tab Replacement Character 15-52
15.6.1.5. Summary of Tabs 15-53
15.6.2. Leaders — Repeated Runs of Characters 15-53
15.6.2.1. **.lc** — Change the Leader Character 15-55
15.6.3. **.fc** — Set Field Characters 15-56
15.7. Titles, Pages, and Numbering 15-60
15.7.1. Three Part Titles for Running Headers and Footers 15-62
15.8. Input and Output when using troff 15-64
15.8.1. **.so** — Read Text from a File 15-64
15.8.2. **.rd** — Read from the Standard Input 15-66
15.8.3. **.tm** — Send Messages to the Standard Error File 15-67
15.9. Using Strings as Shorthand 15-69
15.9.1. **.ds** — Define Strings 15-69
15.9.2. **.as** — Append to a String 15-70
15.10. Macros, Diversions, and Traps 15-73
15.10.1. Macros 15-73
15.10.1.1. **.de** — Define a Macro 15-73
15.10.1.2. Macros with Arguments 15-75
15.10.1.3. **.am** — Append to a Macro 15-76
15.10.1.4. **.rm** — Remove Requests, Macros, or Strings 15-76
15.10.1.5. **.rn** — Rename Requests, Macros or Strings 15-77
15.10.1.6. Copy Mode Input Interpretation 15-79
15.10.2. Using Diversions to Store Text for Later Processing 15-79
15.10.2.1. **.di** — Divert Text 15-79
15.10.2.2. **.da** — Append to a Diversion 15-81
15.10.3. Using Traps to Process Text at Specific Places on a Page 15-81
15.10.3.1. **.wh** — Set Page or Position Traps 15-82
15.10.3.2. **.ch** — Change Position of a Page Trap 15-82
15.10.3.3. **.dt** — Set a Diversion Trap 15-82
15.10.3.4. **.it** — Set an Input-Line Count Trap 15-83
15.10.3.5. **.em** — Set the End of Processing Trap 15-83
15.11. Number Registers and Arithmetic 15-84
15.11.1. **.nr** — Set Number Registers 15-84
15.11.1.1. Auto-increment Number Registers 15-85
15.11.2. Arithmetic Expressions with Number Registers 15-86
15.11.3. **.af** — Specify Format of Number Registers 15-88
15.11.4. **.rr** — Remove Number Registers 15-89
15.12. Arbitrary Motions and Drawing Lines and Characters 15-90

15.12.1. **\u** and **\d** Functions — Half-Line Vertical Movements 15-90
15.12.2. Arbitrary Local Horizontal and Vertical Motions 15-91
15.12.2.1. **\v** Function — Arbitrary Vertical Motion 15-91
15.12.2.2. **\h** Function — Arbitrary Horizontal Motion 15-92
15.12.3. **\0** Function — Digit Sized Spaces 15-93
15.12.4. '**** ' Function — Unpaddable Space 15-94
15.12.5. **\|** and **\^** Functions — Thick and Thin Spaces 15-94
15.12.6. **\&** Function — Non-Printing Zero-Width Character 15-95
15.12.7. **\o** Function — Overstriking Characters 15-96
15.12.8. **\z** Function — Zero Motion Characters 15-97
15.12.9. **\w** Function — Get Width of a String 15-98
15.12.10. **\k** Function — Mark Current Horizontal Place 15-99
15.12.11. **\b** Function — Build Large Brackets 15-99
15.12.12. **\r** Function — Reverse Vertical Motions 15-101
15.12.13. Drawing Horizontal and Vertical Lines 15-101
15.12.13.1. **\l** Function — Draw Horizontal Lines 15-101
15.12.13.2. **\L** Function — Draw Vertical Lines 15-102
15.12.13.3. Combining the Horizontal and Vertical Line Drawing Functions 15-102
15.12.14. **.mc** — Place Characters in the Margin 15-103
15.13. Input and Output Conventions and Character Translations 15-104
15.13.1. Input Character Translations 15-104
15.13.2. **.ec** and **.eo** — Set Escape Character or Stop Escapes 15-104
15.13.3. **.cc** and **.c2** — Set Control Characters 15-104
15.13.4. **.tr** — Output Translation 15-105
15.14. Automatic Line Numbering 15-106
15.14.1. **.nm** — Number Output Lines 15-106
15.14.2. **.nn** — Stop Numbering Lines 15-106
15.15. Conditional Processing of Input 15-108
15.15.1. **.ig** — Ignore Input Text 15-110
15.16. Requests for Debugging your troff Input File 15-113
15.16.1. **.pm** — Display Names and Sizes of Defined Macros 15-113
15.16.2. **.fl** — Flush Output Buffer 15-113
15.16.3. **.ab** — Abort 15-114
15.17. Saving State with Environments 15-115

Appendix A Examples of Fonts and Non—ASCII Characters **A-1**

Appendix B *troff* Request Summary **B-1**

Appendix C Escape Sequences for Characters, Indicators, and Functions **C-1**

Appendix D Predefined Number Registers **D-1**

Appendix E Description of *troff* Output Codes **E-1**

E.1. Codes 00xxxxxx — Flash Codes to Expose Characters E-2
E.2. Codes 1xxxxxxx — Escape Codes Specifying Horizontal Motion E-2
E.3. Codes 011xxxxx — Lead Codes Specifying Vertical Motion E-3
E.4. Codes 0101xxxx — Size Change Codes E-3
E.5. Codes 0100xxxx — Control Codes E-4
E.6. How Fonts are Selected E-5
E.7. Initial State of the C/A/T E-5

Figures

Figure 15-1 Filling and Adjusting Styles 15-16
Figure 15-2 Layout of a Page 15-26

Tables

Table 1-1 Utilities and Their Metacharacters 1-4
Table 2-1 Editor Options 2-13
Table 2-2 File Manipulation Commands 2-20
Table 2-3 Extended Pattern Matching Characters 2-23
Table 2-4 Input Mode Corrections 2-23
Table 2-5 Common Character Abbreviations 2-25
Table 2-6 Terminal Types 2-45
Table 2-7 Frequently-Used `vi` Commands 2-49

Table 7-1 Types of Paragraphs 7-7
Table 7-2 How to Display and Print Documents 7-9
Table 7-3 Registers You Can Change 7-17
Table 8-1 Display Macros 8-13
Table 8-2 Old Accent Marks 8-17
Table 8-3 Accent Marks 8-17
Table 8-4 Units of Measurement in `nroff` and `troff` 8-19
Table 8-5 Summary of **−ms** Number Registers 8-20
Table 8-6 Bell Laboratories Macros (deleted from **−ms**) 8-24
Table 8-7 New **−ms** Requests 8-24
Table 8-8 New String Definitions 8-25
Table 8-9 **−ms** Macro Request Summary 8-25
Table 8-10 **−ms** String Definitions 8-27
Table 8-11 Printing and Displaying Documents 8-28
Table 10-1 *tbl* Command Characters and Words 10-21
Table 11-1 PIC Objects and their Standard Sizes 11-5
Table 12-1 Character Sequence Transalation 12-19
Table 12-2 Greek Letters 12-20
Table 12-3 *eqn* Keywords 12-21
Table 14-1 Special Characters and Diacritical Marks 14-27
Table 14-2 **−me** Request Summary 14-28
Table 15-1 Scale Indicators for Numerical Input 15-9
Table 15-2 Arithmetic Operators and Logical Operators for Expressions 15-10
Table 15-3 Constructs that Break the Filling Process 15-13
Table 15-4 Formatter Requests that Cause a Line Break 15-14
Table 15-5 Adjusting Styles for Filled Text 15-15
Table 15-6 Exceptions to the Standard ASCII Character Mapping 15-46
Table 15-7 Types of Tab Stops 15-53
Table 15-8 Requests that Cause a Line Break 15-61
Table 15-9 Access Sequences for Auto-incrementing Number Registers 15-86
Table 15-10 Arithmetic Operators and Logical Operators for Expressions 15-87
Table 15-11 Interpolation Formats for Number Registers 15-89
Table 15-12 Pieces for Constructing Large Brackets 15-100
Table 15-13 Built In Condition Names for Conditional Processing 15-110
Table B-1 Notes in the Tables B-6
Table D-1 General Number Registers D-1
Table D-2 Read-Only Number Registers D-1

Table E-1 Size Change Codes E-3
Table E-2 C/A/T Control Codes and their Meanings E-4
Table E-3 Correspondence Between Rail, Mag, Tilt, and Font Number E-5

Preface

Editing and Text Processing on the Sun Workstation provides user's guides and reference information for the text editors and document processing tools. We assume you are familiar with a terminal keyboard and the Sun system. If you are not, see the *Beginner's Guide to the Sun Workstation* for information on the basics, like logging in and the Sun file system. If you are not familiar with a text editor or document processor in general, read "An Introduction to Text Editing" and "An Introduction to Document Preparation" in this manual for descriptions of the basic concepts and some simple examples that you can try. Finally, we assume that you are using a Sun Workstation, although specific terminal information is also provided.

If you choose to read one of the user's guides, sit down at your workstation and try the exercises and examples. The reference sections provide additional explanations and examples on how to use certain facilities and can be dipped into as necessary. For additional details on Sun system commands and programs, see the *Commands Reference Manual for the Sun Workstation*.

Summary of Contents

Use the table of contents to Part One and Part Two as roadmaps to guide you to the information you need.

Part One of this manual provides information on the text editors and Part Two describes the document formatting tools.

The contents of Part One are:

1. *An Introduction to Text Editing* — Describes the basics of text editing and provides a guide to the available editing tools. Newcomers should start here.
2. *Using* `vi`, *the Visual Display Editor* — Tutorial and reference information on the visual display editor `vi`. Includes a quick reference to tape up by your workstation.
3. *Command Reference for the* `ex` *Line Editor* — A command reference for the `ex` and `vi` editors. Also includes a quick reference.
4. *Using the* `ed` *Line Editor* — Provides a user's guide to the `ed` tools.
5. *Using* `sed`, *the Stream Text Editor* — A user's guide to `sed`, the non-interactive variant of `ed` for processing large files.
6. *Pattern Scanning and Processing with* `awk` — A user's guide to the `awk` programming language for data transformation and selection operations.

Part Two contains the following chapters:

1. *Introduction to Document Preparation* — Describes the basics of text processing, macros and macro packages, provides a guide to the available tools and several simple examples after which to pattern your papers and documents. Newcomers to the Sun document formatters should start here.
2. *Formatting Documents with the* **−ms** *Macros* — User's guide and reference information for the **−ms** macros for formatting papers and documents. Includes new **−ms** macros.
3. *Formatting Tables with* `tbl` — A user's guide and numerous examples to the table processing utility `tbl`.
4. *PIC — A Graphics Language for Typesetting* — A user's guide and reference guide to the `pic` language for drawing simple pictures.
5. *Typesetting Mathematics with* `eqn` — A user's guide to the `eqn` mathematical equation processor.
6. *Refer — a Bibliography System* — Explains how to use the bibliographic citation program `refer`.
7. *Formatting Documents with the* **−me** *Macros* — Describes the **−me** macro package for producing papers and documents.
8. *Formatting Documents with* `nroff` *and* `troff` — Provides a user's guide and reference material for the `nroff` and `troff` text processors.
9. *The* **−man** Macro Package — Describes the **−man** macro package for producing manual pages.
10. *Examples of Fonts and Non-ASCII Characters* — Several tables demonstrating `troff` fonts and escape sequences for generating non-ASCII characters.
11. `troff` Request Summary — A table summarizing all of the `troff` requests and their arguments.
12. *Escape Sequences for Characters, Indicators, and Functions* — A table summarizing `troff` escape sequences — character sequences beginning with a backslash (\).
13. *Predefined Number Registers* — Tables of `troff` General and Predefined Number Registers
14. Description of `troff`'s Output Codes — A summary of the binary codes for the C/A/T phototypesetter.

Conventions Used in This Manual

Throughout this manual we use

```
hostname%
```

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

Chapter 1

An Introduction to Text Editing

An *editor* is a utility program that you use to modify the contents of a file. A *text* editor deals with files containing a *string* of characters in a particular character set. A *string* is a sequence of characters, 'ABC,' 'evan' or 'm3154' for example. You usually use an editor *interactively*, that is, you can see on the workstation screen what you have and then make changes accordingly.

With a text editor, you can browse through a file, make changes, and then make the changes permanent.

There are also utility programs such as `awk`, `grep`, `fgrep`, `egrep`, and `tr` that operate on a file, but do not change the original file. Rather they modify the data contained in it as the data goes from the original file to the workstation screen, printer, or whatever. Moreover, these commands operate on a global basis, that is, they change everything that conforms to a specific regular pattern. See "Pattern Scanning and Processing with `awk`" in this manual for more information and the *Commands Reference Manual for the Sun Workstation* for details on the other utilities.

There are two kinds of editors, *line* editors and *screen* editors. A *line* editor has a line as its basic unit for change. A line is a string of characters terminated by a newline character, the character that is generated when you type RETURN. You can give the editor commands to do operations on lines, display, change, delete, move, copy a line, or insert a new line. You can substitute character strings within a line or group of lines.

A *screen* editor displays a portion of a file on the workstation screen. You can move the cursor around the screen to indicate where you want to make changes, and you can choose which part of the file to display. Screen editors, such as `vi`, are also called *display* editors.

1.1. Sun System Editors

The Sun system has two basic editors. `ed` is the basic, interactive line editor from which the others have been developed. As they are all related, you can see similarities with `vi`, `ex`, and `sed`. Your primary interface to the Sun system is probably `vi` for editing both source code and text. See the chapter "Using the `ed` Line Editor" for details on `ed`.

The other basic editor is the *stream* editor `sed`, which as a lineal descendant of `ed`, can perform similar operations. However, it is not interactive and you cannot move backwards in the edit file. You specify the command or series of commands to be executed, and `sed` performs them from the beginning to the end of the file. Because `sed` does not copy your file into the buffer to create a temporary file like `ed` does, you can use `sed` to edit any size file. `sed` is usually used for making transient changes only. `sed` recognizes basically the same *regular expressions* as `ed`. Regular expressions are described below. See the chapter "Using `sed`, the Stream Text Editor" for instructions on how to use `sed`.

More useful for general text editing are the screen editors **ex** and **vi**. A variant of **ex**, **edit**, has features designed to make it less complicated to learn and use.[1]

ex is also based on **ed**, but has many extensions and additional features. Commands are less cryptic and hence, easier to remember. There are variants of some editor operations, which modify the way in which those operations are performed under certain conditions. **ex** is more communicative, displaying more descriptive error messages than merely '?' as **ed** does and providing instructions on how to override the error condition. There are editor ***options*** which modify overall **ex** behavior. **ex** also provides the ***visual*** mode, which turns **ex** into a screen editor. In this mode, **ex** is identical to **vi**. You can use the ***open*** mode for intraline editing.

vi is the ***screen***, ***display***, or ***visual*** editor version of **ex**. A portion of the file you wish to modify is displayed on your workstation screen. Within the displayed portion of the file, you can move the cursor around to control where changes are to be made, and then you can make changes by replacing, adding or deleting text. You can change the portion of the file displayed on the screen, so you have access to the whole file.

You also have access to all of the **ex** line-oriented commands from **vi**. Many of the more useful operations that can be performed in **vi** simply call upon **ex** functions. Additionally, some operations, such as global substitutions, are easily performed using **ex** from **vi**. Because of this connection, refer to both chapters: "Using **vi**, the Visual Display Editor" and the "Command Reference for the **ex** Line Editor". For a quick tutorial on the most useful **vi** commands and features, read the chapter on **vi** in the ***Beginner's Guide to the Sun Workstation***.

1.2. Text Editing Basics

In editing jargon, we say you ***enter*** an editor to edit a file and ***quit*** an editor to return to the system command level shell.

Most editors set aside a temporary work space, called a ***buffer***, separate from your permanent file. Before starting to work on an existing file, the editor makes a copy of it in the buffer, leaving the original untouched. When you make editing changes to the buffer copy, you must then save or ***write*** the file to make the changes permanent. The buffer disappears at the end of the editing session.

During an editing session there are two usual modes of operation: ***command*** mode and ***text input*** mode. (This disregards, for the moment, ***open*** and ***visual*** modes, discussed below.) In command mode, the editor may prompt you with a question mark (?), a colon (:), or nothing at all as in **vi**. In text input mode, there is no prompt and the editor merely adds the text you type in to the buffer. You start text input mode with a command that ***appends***, ***inserts***, or ***changes***, and terminate it either by typing a period as the first and only character on a line for **ed** and **ex** or by typing the ESCAPE (ESC) key for **vi**.

The editor keeps track of lines of text in the buffer by numbering them consecutively starting with 1 and renumbering as lines are added or deleted. It doesn't normally display the line numbers, although you can specify that they be displayed in **vi**. At any given time the editor is positioned at one of these lines; this position is called the ***current line***.

Some editor commands take line-number prefixes. The concept of line numbers is especially important in **ed** and **ex**; you use them to indicate which lines to operate on. You also use line

[1] See *Edit: A Tutorial*, Ricki Blau and James Joyce, University of California, Berkeley.

numbers in `vi`, but less frequently. With `ed`, you can precede most commands by one or two line-number addresses which indicate the lines to be affected. If you give one line number, the command operates on that line only; if you give two, it operates on an inclusive range of lines. Commands that can take line-number prefixes also assume default prefixes if none are given. The default assumed by each command is designed to make it convenient to use in many instances without any line-number prefix. For the most part, a command used without a prefix operates on the current line, though there are exceptions to this rule. The `print` command by itself, for instance, displays one line, the current line, on the workstation screen. In the address prefix notation, '.' stands for the current line and '$' stands for the last line of the buffer. If no such notation appears, no line-number prefix may be used. Some commands take trailing information.

Besides command and text input modes, `ex`, `vi`, and `edit` provide other modes of editing called ***open*** and ***visual***. In these modes you can move the cursor to individual words or characters in a line. The commands you then use are very different from the standard editor commands; most do not appear on the screen when typed.

1.2.1. Regular Expressions in Text Patterns

You can use the editors and the utilities mentioned above to deal with fixed strings of characters, but this may become tedious if you want to do something more complex. You can also specify a pattern or template of text you want to modify; this pattern is called a ***regular expression***. Certain characters take on special meanings when used in in these patterns. These special characters are called ***metacharacters*** because they represent something other than themselves.

Here is a table of all the special metacharacters and which utilities support those particular characters.

Table 1-1: Utilities and Their Metacharacters

Character	Meaning	Supported by						
		fgrep	**grep**	**egrep**	**awk**	**ed**	**sed**	**ex**
c	any character except specials	yes	yes	yes	yes	yes	yes	yes
^	match beginning of line	yes	yes	yes	yes	yes	yes	yes
$	match end of line	yes	yes	yes	yes	yes	yes	yes
\<	match beginning of word	no	yes	no	no	no	no	yes
\>	match end of word	no	yes	no	no	no	no	yes
.	any character	no	yes	yes	yes	yes	yes	yes
[*string*]	character class	no	yes	yes	yes	yes	yes	yes
[^*string*]	negated character class	no	yes	yes	yes	yes	yes	yes
*	closure	no	yes	yes	yes	yes	yes	yes
(*pattern*)	grouping	no	no	yes	yes	no	no	no
|	alternation	no	no	yes	yes	no	no	no
\(*pattern*\)	remember *pattern*	no	no	no	no	yes	yes	yes

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

Some of the metacharacters that **ed** and some of the other utilities use are also used by the shell for matching filenames, so you should enclose the regular expression in single quotes (').

You can combine regular expressions to specify a lot more than just a single string of text, so you can give the editor commands that operate on either a very specific string of text or ***globally*** on a whole file.

See the ***Beginner's Guide to the Sun Workstation*** for a more detailed and descriptive explanation of regular expressions.

1.3. What to Do If Something Goes Wrong

Sometimes you may make a mistake or your system may not respond correctly. Here are some suggestions on what to do.

If you make a mistake in the editor that you cannot recover from easily, do not panic. As long as you do not ***write*** the file ***and*** quit the editor, you can retrieve the original file. Force the editor to quit (in `vi`, for example, you type `:q!`, the exclamation point overriding any warning), and then enter the editor again to start over. When you try to quit the editor without saving changes, the editor will warn you that you have unsaved changes, so you have to force the quit with '!'.

At the Sun system level, if you make a typing mistake, and see it before you press RETURN, there are several ways to recover. The DEL key is the ***erase*** character. Use it to back up over and erase the previously-typed character. Use the DEL key repeatedly to erase characters back to the beginning of the line, but not beyond. Use CTRL[2] to abort or ***send an interrupt*** to a currently running program. You can't interrupt an editor with CTRL-C.

Sometimes you can get into a state where your workstation or terminal acts strangely. For example, you may not be able to move the cursor, your cursor may disappear, there is no echoing of what you type, or typing RETURN may not cause a linefeed or return the cursor to the left margin. Try the following solutions:

- First, type CTRL-Q to resume possibly suspended output. (You might have typed CTRL-S, freezing the screen.)
- Another possibility is that you accidentally typed a NO SCRL key (also called SET UP/NO SCROLL on some terminals) on your keyboard. This also freezes the keyboard like typing a CTRL-S. Try typing CTRL-Q, which toggles you back to proper operation if you did indeed type the NO SCRL key in the first place.
- Next, try pressing the LINEFEED key, followed by typing RESET, and pressing LINEFEED again.
- If that doesn't help, try logging out and logging back in. If you are using a terminal, try powering it off and on to regain normal operation.
- If you get unwanted messages or garbage on your screen, type CTRL-L to refresh the workstation screen. (Use CTRL-R on a terminal.)

If your system goes down, a file with almost all of your latest changes is automatically saved. After rebooting your system, or doing whatever needs to be done, you will receive mail indicating that the file has been saved. First, return to the directory where the file belongs, and then re-enter the editor with the `-r` option to ***restore*** the file:

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

This returns you to a version of the file you were editing, minus a few of your most recent changes.

---

[2] We use the convention CTRL-*whatever* to mean you hold down the control (or CTRL) key while typing a *whatever* character. CTRL-C means hold down the CONTROL key while typing 'c'. The case does not matter; CTRL-C and CTRL-c are equivalent.

Chapter 2

Using vi, the Visual Display Editor

This chapter[3] describes vi (pronounced *vee-eye*) the visual, display editor. The first part of this chapter provides the basics of using vi. The second part provides a command reference and terminal set-up information. Finally, there is a quick reference, summarizing the vi commands. Keep this reference handy while you are learning vi. As the vi editor is the visual display version of the ex line editor, and because the full command set of the line-oriented ex editor is available within vi, you can use the ex commands in vi. Some editing, such as global substitution, is more easily done with ex. So refer to the information in the chapter "Command Reference for the ex Line Editor" as it also applies to vi.

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

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

2.1. vi and ex

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

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

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

---

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

2.2. Getting Started

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

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

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

2.2.1. Editing a File

To use vi on the file, type:

```
hostname% vi filename
```

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

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

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

2.2.2. The Editor's Copy — Editing in the Buffer

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

2.2.3. *Arrow Keys*

The editor command set is independent of the workstation or terminal you are using. On most terminals with cursor positioning keys, these keys will also work within the editor.[4] If you don't have cursor positioning keys, that is, keys with arrows on them, or even if you do, you can use the `h`, `j`, `k`, and `l` keys as cursor positioning keys. As you will see later, `h` moves back to the left (like CTRL-H, a backspace), `j` moves down (in the same column), `k` moves up (in the same column), and `l` moves the cursor to the right.

2.2.4. *Special Characters: ESC, CR and CTRL-C*

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

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

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

Try typing the '/' key on your keyboard. Use this key to search for a string of characters. `vi` displays the cursor at the bottom line of the screen after a '/' is displayed as a prompt. You can get the cursor back to the current position by pressing BACK SPACE (or DEL); try this now. This cancels the search. Typing CTRL-C also cancels the search. From now on we will simply refer to typing CTRL-C (or pressing the DEL or RUB key) as 'sending an interrupt.'[5]

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

2.2.5. *Getting Out of* `vi` — `:q`, `:q!`, `:w`, `ZZ`, `:wq`

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

---

[4] Note for the HP2621: on this terminal the function keys must be ***shifted*** to send to the machine, otherwise they only act locally. Unshifted use leaves the cursor positioned incorrectly.

[5] On some systems, this interruptibility comes at a price: you cannot type ahead when the editor is computing with the cursor on the bottom line.

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

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

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

2.3. Moving Around in the File

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

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

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

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

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

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

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

2.3.2. Searching, Goto, and Previous Context — /, ?, G

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

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

```
/today/;/tomorrow
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.3.3. Moving Around on the Screen — h, j, k, l

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

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

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

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

2.3.4. Moving Within a Line — b, w, e, E, B, W

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

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

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

2.3.5. Viewing a File — view

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

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

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

2.4. Making Simple Changes

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

2.4.1. Inserting — `i` *and* `a`

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

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

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

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

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

Whenever you are typing in text, you can give many lines of input or just a few characters. To type in more than one line of text, type a RETURN at the middle of your input. A new line will be created for text, and you can continue to type. (If you are on a slow, dumb terminal `vi` may choose to wait to redraw the tail of the screen, and will let you type over the existing screen lines. This avoids the lengthy delay that would occur if `vi` attempted to always keep the tail of the screen up to date. The tail of the screen will be fixed up, and the missing lines will reappear, when you type ESC.)

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

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

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

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

2.4.2. Making Small Corrections — x, r, s, R

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

If the character is incorrect, you can ***replace*** it with the correct character by typing the command *rc*, where *c* is replaced by the correct character. You don't need to type ESC. Finally if the character that is incorrect should be replaced by more than one character, type s which ***substitutes*** a string of characters, ending with ESC, for it. If there are a small number of characters that are wrong you can precede s with a count of the number of characters to be replaced. You can use counts with x to specify the number of characters to be deleted and with r, such as 4rx to specify that a character be replaced with four x's.

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

2.4.3. Deleting, Repeating, and Changing — dw, ., db, c

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

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

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

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

2.4.4. Operating on Lines — dd, cc, S

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

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

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

You can delete or change more than one line by preceding the dd or cc with a count, such as 5dd, which deletes 5 lines. You can also give a command like dL to delete all the lines up to and including the ***last*** line on the screen, or d3L to delete through the third from the bottom line. Try some commands like this now.[6] Notice that vi lets you know when you change a large number of lines so that you can see the extent of the change. It also always tells you when a change you make affects text you cannot see.

2.4.5. Undoing — u, U

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

The undo command lets you reverse only a single change. After you make a number of changes to a line, you may decide that you would rather have the original state of the line back. The U command restores the current line to the state before you started changing it only while the cursor is still on that line. If you move the cursor away from the line you changed, U does nothing.

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

---

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

2.5. Moving About: Rearranging and Duplicating Text

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

2.5.1. Low-level Character Motions — f, F, ^

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

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

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

Your file may have tab (CTRL-I) characters in it. These characters are represented as a number of spaces expanding to a tab stop, where tab stops are every eight positions.[7] When the cursor is at a tab, it sits on the last of the several spaces that represent that tab. Try moving the cursor back and forth over tabs so you understand how this works.

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

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

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

In working with a document it is often advantageous to work in terms of sentences, paragraphs, and sections. The operations '(' and ')' move to the beginning of the previous and next sentences respectively. Thus the command d) deletes the rest of the current sentence; likewise d(deletes the previous sentence if you are at the beginning of the current sentence, or the current

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

sentence up to where you are if you are not at the beginning of the current sentence.

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

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

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

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

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

2.5.3. Rearranging and Duplicating Text — y, Y, p, P

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

The operator y ***yanks*** a copy of the object that follows into the unnamed buffer. If preceded by a buffer name, "*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 you yank forms a part of a line, or is an object such as a sentence that partially spans more than one line, then when you put the text back, it will be placed after the cursor (or before if you use P). If the yanked text forms whole lines, they will be put back as whole lines, without changing the current line. In this case, the put acts much like an o or O command.

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

To move text within the buffer, you need to delete it in one place, and put it back in another. You can precede a delete operation by the name of a buffer in which the text is to be stored as

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

2.6. High-Level Commands

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

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

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

:q!<CR>

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

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

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

2.6.2. Escaping to a Shell — :!, :sh, CTRL-Z

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

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

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

2.6.3. Marking and Returning — m

The command `` returned to the previous place after a motion of the cursor by a command such as /, ? or G. You can also *mark* lines in the file with single letter tags and return to these marks later by naming the tags. Try marking the current line with the command 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 type `a. The cursor will return to the place you marked. Marks last only until you edit another file.

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

2.6.4. Adjusting the Screen CTRL-L, z

If the screen image is messed up because of a transmission error to your workstation, or because some program other than vi wrote output to your workstation, you can type a CTRL-L, the ASCII form-feed character, to refresh the screen. (On a dumb terminal, if there are @ lines in the middle of the screen as a result of line deletion, you may get rid of these lines by typing CTRL-R to *retype* the screen, closing up these holes.[8])

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

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

2.7. Special Topics

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

2.7.1. Options, the Set Variable, and Editor Start-up Files

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

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:

[8] This includes Televideo 912/920 and ADM31 terminals.

Table 2-1: Editor Options

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

```
set opt=val
```

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

```
set opt
set noopt
```

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

```
:se nu
```

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

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

or the value of a single option with

```
:set opt?<CR>

:set noai?<CR>
noautoindent
```

The

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

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

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

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

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

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

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

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

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

2.7.2. Recovering Lost Lines

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

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

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

2.7.3. Recovering Lost Files — the −r Option

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

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

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

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

```
hostname% vi -r
```

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

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

2.7.4. Continuous Text Input — wrapmargin

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

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

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

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

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

2.7.5. Features for Editing Programs

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

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

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

After you have started a new line, you might want to indent your current line less than the previous line. You are still in insert mode, and cannot backspace over the automatic indentation. However, you can type CTRL-D to backtab over each level of indentation. Each time you type CTRL-D, you back up one position, normally to an eight-column boundary. You can set the number of columns that a tab shifts with the ***shiftwidth*** option. Try giving the command:

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

and then experimenting with autoindent again.

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

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

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

2.7.6. *Filtering Portions of the Buffer*

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

```
!}sort<CR>
```

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

2.7.7. *Commands for Editing LISP*

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

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

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

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

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

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

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

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

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

2.7.8. *Macros*

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

Briefly, there are two kinds of macros:

1. Ones where you put the macro body in a buffer register, say *x*. You can then type **@x** to invoke the macro. The **@** may be followed by another **@** to repeat the last macro.

2. You can use the **map** command from **vi** (typically in your EXINIT) with a command of the form:

   :map *lhs rhs*<CR>[9]

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

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

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

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

You can delete macros with

```
:unmap lhs
```

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

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

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

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

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

```
:map! ^T ^V␣␣␣␣
```

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

```
:map!
```

to list macros that apply during input mode and

```
:map
```

to list macros that apply during command mode.

---

[9] *lhs* is an abbreviation for *left hand side*. *rhs* is an abbreviation for *right hand side*.

2.7.9. Word Abbreviations — :ab, :una

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

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

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

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

```
:una foo
```

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

2.8. Nitty-gritty Details

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

2.8.1. Line Representation in the Display

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

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

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

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

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

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

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

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

To turn this off, use:

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

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

2.8.2. Command Counts

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

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

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

vi uses the default window size when it clears and refills the screen after a search or other motion moves far from the edge of the current window. All commands that take a new window size as count often redraw the screen. If you anticipate this, but do not need as large a window as you are currently using, you may wish to change the screen size by specifying the new size before these commands. In any case, the number of lines used on the screen will expand if you move off the top with a '–' or similar command or off the bottom with a command such as RETURN or CTRL-D. The window will revert to the last specified size the next time it is cleared and refilled, but not by a CTRL-L which just redraws the screen as it is.

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

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

2.8.3. File Manipulation Commands

The following table lists the file manipulation commands you can use when you are in vi. A CR or ESC follows all of these commands. The most basic commands are :w and :e. End a normal editing session on a single file with a ZZ command. If you are editing for a long period of time, use the :w command occasionally after major amounts of editing, and then finish with a ZZ. When you edit more than one file, you can finish with one with a :w and start editing a new file

Table 2-2: File Manipulation Commands

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

by giving a `:e` command, or set *autowrite* and use `:n` *file*.

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

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

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

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

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

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

same tag again.

2.8.4. *More about Searching for Strings*

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

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

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

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

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

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

```
set nomagic
```

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

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

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

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

Table 2-3: Extended Pattern Matching Characters

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

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

2.8.5. More about Input Mode

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

Table 2-4: Input Mode Corrections

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

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

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

If you wish to type in your erase or kill character, say CTRL-U, you must precede it with a \, just as you would do at the normal system command level. A more general way of typing non-printing characters into the file is to precede them with a CTRL-V. The CTRL-V echoes as a ↑

character on which the cursor rests. This indicates that the editor expects you to type a control character. In fact you may type any character and it will be inserted into the file at that point.[10]

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

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

2.9. Command and Function Reference

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

2.9.1. Notation

Notation used in this section is as follows.

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

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

{*variable item*}
Denotes parts of the command that must appear, but can take a number of different values.

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

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

Table 2-5: Common Character Abbreviations

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

2.9.2. Commands

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

2.9.3. Entry and Exit

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

```
hostname% vi filename
```

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

To exit from `vi`, type:

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

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

2.9.4. Cursor and Page Motion

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

[count]<bs> or [count]h or [count]←
: Move the cursor to the left one character. Cursor stops at the left margin of the page. [count] specifies the number of spaces to move.

[count]CTRL-N or [count]j or [count]↓ or [count]<lf>
: Move down one line. Moving off the screen scrolls the window to force a new line onto the screen. Mnemonic: **N**ext

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

[count]<sp> or [count]l or [count]→
Move to the right one character. Cursor will not go beyond the end of the line.

[count]– Move the cursor up the screen to the beginning of the next line. Scroll if necessary.

[count]+ or [count]CR
Move the cursor down the screen to the beginning of the next line. Scroll up if necessary.

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

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

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

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

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

[count]W Move the cursor to the beginning of the next word that follows a blank space (<sp>,<tab>, or <nl>). Ignore other punctuation.

[count]b Move the cursor to the preceding word. Mnemonic: **b**ackup-word

[count]B Move the cursor to the preceding word that is separated from the current word by a blank space (<sp>,<tab>, or <nl>).

[count]e Move the cursor to the end of the current word or the end of the ***count***th word hence. Mnemonic: **e**nd-of-word

[count]E Move the cursor to the end of the current word which is delimited by blank space (<sp>,<tab>, or <nl>).

[line number]G
Move the cursor to the line specified. Of particular use are the sequences 1G and G, which move the cursor to the beginning and the end of the file respectively. Mnemonic: **G**o-to

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

[count]CTRL-D Move the cursor down in the file by ***count*** lines (or the last ***count*** if a new count isn't given). The initial default is half a page. The screen is simultaneously scrolled up. Mnemonic: **D**own

[count]CTRL-U Move the cursor up in the file by ***count*** lines. The screen is simultaneously scrolled down. Mnemonic: **U**p

[count]CTRL-F Move the cursor to the next page. A count moves that many pages. Two lines of the previous page are kept on the screen for continuity if possible.

Mnemonic: **F**orward

[count]CTRL-B Move the cursor to the previous page. Two lines of the current page are kept if possible. Mnemonic: **B**ackward

[count]) Move the cursor to the beginning of the next sentence. A sentence is defined as ending with a '.', '!', or '?' followed by two spaces or a <nl>.

[count](Move the cursor backward to the beginning of a sentence.

[count]} Move the cursor to the beginning of the next paragraph. This command works best inside `nroff` documents. It understands the `nroff` macros in **−ms**, for which the commands `.IP`, `.LP`, `.PP`, `.QP`, 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 are adjustable. See the entry for "Paragraphs" in the "Set Commands" section.

[count]{ Move the cursor backward to the beginning of a paragraph.

]] Move the cursor to the next 'section,' where a section is defined by the set of `nroff` macros in **−ms**, in which `.NH`, `.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 the "sections" entry under the heading "Set Commands".

[[Move the cursor backward to the beginning of a section.

% Move the cursor to the matching parenthesis or brace. This is very useful in C or lisp code. If the cursor is sitting on a `(`, `)`, `{`, or `}`, it is moved to the matching character at the other end of the section. If the cursor is not sitting on a brace or a parenthesis, `vi` searches forward on that line until it finds one and then jumps to the match mate.

[count]H If there is no count, move the cursor to the top left position on the screen. If there is a count, then move the cursor to the beginning of the line ***count*** lines from the top of the screen. Mnemonic: **H**ome

[count]L If there is no count, move the cursor to the beginning of the last line on the screen. If there is a count, move the cursor to the beginning of the line ***count*** lines from the bottom of the screen. Mnemonic: **L**ast

M Move the cursor to the beginning of the middle line on the screen. Mnemonic: **M**iddle

m<a-z> ***Mark*** the place in the file without moving the cursor; use a character from a to z, '<a-z>', as the label for referring to this location in the file. See the next two commands. Mnemonic: **m**ark Note: the ***mark*** command is not a motion and 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 searching or jumping to a specific line in the file, rather than moving the cursor or scrolling the screen.

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

2.9.5. Searches

The following commands search for items in a file.

[count]f{chr} Search forward on the line for the next or ***count***th occurrence of the character ***chr***. The cursor is placed ***at*** the character of interest. Mnemonic: **f**ind character

[count]F{chr} Search backward on the line for the next or ***count***th occurrence of the character ***chr***. The cursor is placed ***at*** the character of interest.

[count]t{chr} Search forward on the line for the next or ***count***th occurrence of the character ***chr***. The cursor is placed ***just preceding*** the character of interest. Mnemonic: move cursor up **t**o character

[count]T{chr} Search backward on the line for the next or ***count***th occurrence of the character ***chr***. The cursor is placed ***just preceding*** the character of interest.

[count]; Repeat the last `f`, `F`, `t` or `T` command in the same search direction.

[count], Repeat the last `f`, `F`, `t` or `T` command, but in the opposite search direction. This is useful if you overshoot what you are looking for.

[count]/[string]/<nl>
Search forward for the next occurrence of 'string'. Wraparound at the end of the file does occur. The final / is not required.

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

n Repeat the last /[string]/ or ?[string]? search. Mnemonic: **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 like you can in the `ed` editor.

2.9.6. Text Insertion

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

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

2.9.7. Text Deletion

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

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

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

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

[count]d{motion}
Delete one or more occurrences of the specified motion. You can use any motion here described in the secions "Low Level Character Motions" and "Higher Level Text Objects" . You can repeat the d (such as [***count***]dd) to delete ***count*** lines.

2.9.8. Text Replacement

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

r<chr> Replace the character at the current cursor position with <chr>. This is a one-character replacement. No ESC is required for termination. Mnemonic: **r**eplace character

R{text}<esc> Start overlaying the characters on the screen with whatever you type. It does not stop until you type an ESC.

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

[count]S{text}<esc>
Substitute for the entire current line or lines. If you do not give a count, a '$' appears at the end of the current line. If you give a count of more than 1, all the lines to be replaced are deleted before the insertion begins.

[count]c{motion}{text}<esc>
Change the specified ***motion*** by replacing it with the insertion text. A '$' appears at the end of the last item that is being deleted unless the deletion

involves whole lines. Motions can be any motion from the sections "Low Level Character Motions" and "Higher Level Text Objects". Repeat the c (such as [*count*]cc) to change ***count*** lines.

2.9.9. Moving Text

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

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

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

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

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

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

[count]>{motion} The shift operator right shifts all the text from the line on which the cursor is located to the line where the ***motion*** is located. The text is shifted by one ***shiftwidth***. (See the "Terminal Information" section.) >> means right shift the current line or lines.

[count]<{motion} The shift operator left shifts all the text from the line on which the cursor is located to the line where the ***item*** is located. The text is shifted by one ***shiftwidth***. (See the section on "Terminal Information".) << means left shift the current line or lines. Once the line has reached the left margin, it is not affected further.

[count]={motion} Prettyprints the indicated area according to LISP conventions. The area should be a LISP s-expression.

2.9.10. Miscellaneous Commands

A number of useful miscellaneous **vi** commands follow:

ZZ — Exit from **vi**. If any changes have been made, the file is written out. Then you are returned to the shell.

CTRL-L — Redraw the current screen. This is useful if messages from a background process are displayed on the screen, if someone 'writes' to you while you are using **vi** or if for any reason garbage gets onto the screen.

CTRL-R — On dumb terminals, those not having the 'delete-line' function (the vt100 for example), **vi** saves redrawing the screen when you delete a line by just marking the line with an '@' at the beginning and blanking the line. If you want to actually get rid of the lines marked with '@' and see what the page looks like, type a CTRL-R.

. — 'Dot' repeats the last text modifying command. You can type a command once and then move to another place and repeat it by just typing '.'.

u — Undo the last command that changed the buffer. Perhaps the most important command in the editor. Mnemonic: **u**ndo

U — Undo all the text modifying commands performed on the current line since the last time you moved onto it.

[count]J — Join the current line and the following line. The <nl> is deleted and the two lines joined, usually with a space between the end of the first line and the beginning of what was the second line. If the first line ended with a 'period', two spaces are inserted. A count joins the next ***count*** lines. Mnemonic: **J**oin lines

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

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

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

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

2.9.11. Special Insert Characters

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

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

[^]CTRL-D CTRL-D without any argument backs up one ***shiftwidth***. Use this to remove indentation that was inserted by the ***autoindent*** feature. Typing ^CTRL-D temporarily removes all the autoindentation, thus placing the cursor at the left margin. On the next line, the previous indent level is restored. This is useful for putting 'labels' at the left margin. 0CTRL-D removes all autoindents and keeps it that way. Thus the cursor moves to the left margin and stays there on successive lines until you type TABs. As with the TAB, the CTRL-D is effective only before you type any other 'non-autoindent' controlling characters. Mnemonic: **D**elete a shiftwidth

CTRL-W If the cursor is sitting on a word, CTRL-W moves the cursor back to the beginning of the word, erasing the word from the insert. Mnemonic: erase **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, quote it with the CTRL-V.

2.9.12. : Commands

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

:w[!] [file] Write out the current text to the disk. It is written to the file you are editing unless you supply ***file***. If ***file*** is supplied, the write is directed to that file instead. If that file already exists, vi does not write unless you use the '!' indicating you ***really*** want to write over the older copy of the file.

:q[!] Exit from vi. If you have modified the file you are currently looking at and haven't written it out, vi refuses to exit unless you type the !.

:e[!] [+[cmd]] [file]
Start editing a new file called ***filename*** 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'. Use it if you really mess up the file. The optional '+' says instead of starting at the beginning, start at the 'end', or, if you supply ***cmd***, execute ***cmd*** first. Use this where ***cmd*** is ***n*** (any integer) that starts at line number ***n***, and ***/text***, searches for 'text' and starts at the line where it is found.

CTRL-^ Switch back to the place in the previous file that you were editing with vi, before you switched to the current file.

:n[!] Start editing the next file in the argument list. Since you can call vi with multiple filenames, the :n command tells it to stop work on the current file and switch to the next file. If you have modified the current file, it has to be written out before the :n will work or else you must use '!', which discards the changes you 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 system shell command.

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

tag ***filename*** vi-search-command

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

2.9.13. Set Commands

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

:set *option*<nl>

and turn off the toggle options with the command:

:set *nooption*<nl>

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

:set *option=value*<nl>

To display the value of a specific option, type:

:set *option?*<nl>

To display only those that you have changed, type:

:set<nl>

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

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

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

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

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

or

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

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

autoindent ai
Default: noai Type: toggle
When in autoindent mode, **vi** helps you indent code by starting each line in the same column as the preceding line. Tabbing to the right with <tab> or CTRL-T moves this boundary to the right; to move it to the left, use CTRL-D.

autoprint ap
Default: ap Type: toggle
Displays the current line after each **ex** text modifying command. Not of much interest in the normal **vi** visual mode.

autowrite aw
Default: noaw type: toggle
Does an automatic write if there are unsaved changes before certain commands that change files or otherwise interact with the outside world are executed. These commands are **:!**, **:tag**, **:next**, **:rewind**, CTRL-^, and CTRL-].

beautify bf
Default: nobf Type: toggle
Discards all control characters except <tab>, <nl>, and <ff>.

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 the Sun system.

ignorecase ic
Default: noic Type: toggle
Map all upper-case characters to lower case in regular expression matching.

lisp
Default: nolisp Type: toggle
Autoindent for LISP code. The commands **(**, **)**, **[[**, and **]]** are modified appropriately to affect s-expressions and functions.

list
Default: nolist Type: toggle
Show the <tab> and <nl> characters visually on all displayed lines.

magic
Default: magic Type: toggle
Enable the metacharacters for matching. These include **.**, *****, **<**, **>**, **[*string*]**, **[^*string*]**, and **[<*chr*>-<*chr*>]**.

number nu
Default: nonu Type: toggle
Display each line with its line number.

open
Default: open Type: toggle
When set, prevents entering open or visual modes from **ex** or **edit**. Not of interest from **vi**.

optimize opt
Default: opt Type: toggle
Useful only when using the **ex** capabilities. This option prevents automatic <cr>s from taking place, and speeds up output of indented lines, at the expense of losing typeahead on some versions of the operating system.

paragraphs para
Default: para=lPLPPPQPP bp Type: string
Each pair of characters in the string indicates **nroff** macros to be treated as the beginning of a paragraph for the { and } commands. The default string is for the **−ms** macros. To indicate one-letter **nroff** macros, such as .P or .H, insert a space for the second character position. For example:

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

causes **vi** to consider .PP, .H and .bp as paragraph delimiters.

prompt
Default: prompt Type: toggle
In **ex** command mode the prompt character : is displayed 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
Set the threshold for the number of lines modified. When more than this number of lines is modified, removed, or yanked, **vi** reports 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 CTRL-U and CTRL-D commands.

sections
Default: sections=SHNHH HU Type: string
Each two-character pair of this string specifies **nroff** macro names that are to be treated as the beginning of a section by the]] and [[commands. The default string is for the **−ms** macros. To enter one-letter **nroff** macros, use a quoted space as the second character. See the "Paragraphs" entry for a fuller explanation.

shell sh
Default: sh=from environment SHELL or /bin/sh Type: string
Specify the name of the ***sh*** to be used for 'escaped' commands.

shiftwidth sw
Default: sw=8 Type: numeric
Specify the number of spaces that a CTRL-T or CTRL-D will move over for indenting, and the amount that < and > will shift by.

showmatch sm
Default: nosm Type: toggle
When a) or } is typed, show the matching (or { by moving the cursor to it for one second if it is on the current screen.

slowopen slow
Default: terminal dependent Type: toggle
Prevent updating the screen some of the time to improve speed on terminals that are slow and dumb.

tabstop ts
Default: ts=8 Type: numeric
<tab>s are expanded to boundaries that are multiples of this value.

taglength tl
Default: tl=0 Type: numeric
If nonzero, tag names are only significant to this many characters.

term
Default: (from environment TERM, else dumb) Type: string
This is the terminal and controls the visual displays. It cannot be changed when in visual mode; you have to type a `Q` to change to command mode, type a `set term` command, and enter `vi` to get back into visual. Or exit from `vi`, fix $TERM, and re-enter. The definitions that drive a particular terminal type are in the file */etc/termcap*.

terse
Default: terse Type: toggle
When set, the error diagnostics are short.

warn
Default: warn Type: toggle
Warns if you try to escape to the shell without writing out the current changes.

window
Default: window={8 at 600 baud or less, 16 at 1200 baud, and screen size − 1 at 2400 baud or more} Type: numeric
Specify the number of lines in the window whenever `vi` must redraw an entire screen. It is useful to make this size smaller if you are on a slow line.

w300, w1200, w9600
Set the window, but only within the corresponding speed ranges. They are useful in an EXINIT to fine tune window sizes. For example,

```
set w300=4 w1200=12
```

produces a four-line window at speeds up to 600 baud, a 12-line window at 1200 baud, and a full-screen window (the default) at over 1200 baud.

wrapscan ws
Default: ws Type: toggle
Searches will wrap around the end of the file when is option is set. When it is off, the search will terminate when it reaches the end or the beginning of the file.

wrapmargin wm
Default: wm=0 Type: numeric
`vi` automatically inserts a <nl> when it finds a natural break point (usually a <sp> between words) that occurs within ***wm*** spaces of the right margin. Therefore with 'wm=0', the option is off. Setting it to 10 means that any time you are within 10 spaces of the right margin, `vi` looks for a <sp> or <tab> that it can replace with a <nl>. This is convenient if you forget to look at the screen while you type. If you go past the margin (even in the middle of a word), the entire word is erased and rewritten on the next line.

writeany wa
Default: nowa Type: toggle
`vi` normally makes a number of checks before it writes out a file. This prevents you from inadvertently destroying a file. When the ***writeany*** option is enabled, `vi` no longer makes these checks.

2.9.14. Character Functions

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

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

CTRL-@ Not a command character. If typed as the first character of an insertion, it is replaced with the last text inserted, and the insert terminates. Only 128 characters are saved from the last insert; if more characters were inserted the mechanism is not available. A CTRL-@ cannot be part of the file due to the editor implementation.

CTRL-A Unused.

CTRL-B Scroll backward one window. A count specifies repetition. The top two lines in the window before typing CTRL-B appear as the bottom two lines of the next window.

CTRL-C Unused.

CTRL-D As a command, scrolls down a half window of text. A count gives the number of (logical) lines to scroll, and is remembered for future CTRL-D and CTRL-U commands. During an insert, CTRL-D backtabs over ***autoindent*** blank space at the beginning of a line. This blank space cannot be backspaced over.

CTRL-E Exposes one more line below the current screen in the file, leaving the cursor where it is if possible.

CTRL-F Move forward one window. A count specifies repetition. The bottom two lines in the window before typing CTRL-F appear as the top two lines of the next window.

CTRL-G Equivalent to `:f`CR. These commands display the current file, a message if the file has been modified, the line number of the line the cursor is on, the total number of lines in the file, and the percentage of the way through the file that the current line is.

CTRL-H (BS) Same as ← (see `h`). During an insert, CTRL-H eliminates the last input character, backing over it but not erasing it; the character remains so you can see what you typed if you wish to type something only slightly different.

CTRL-I (TAB) Not a command character. When inserted it prints as some number of spaces. When the cursor is at a tab character, it rests at the last of the spaces that represent the tab. The ***tabstop*** option controls the spacing of tabstops.

CTRL-J (LF) Same as ↓ (see `j`).

CTRL-K Unused.

CTRL-L The ASCII formfeed character, that clears and redraws the screen. This is useful after a transmission error, if characters typed by a program other than the editor scramble the screen, or after output is stopped by an interrupt.

CTRL-M (CR) A carriage return advances to the next line, at the first non-blank position in the line. Given a count, it advances that many lines. During an insert, a CR causes the insert to continue onto another line.

CTRL-N Same as ↓ (see j).

CTRL-O Unused.

CTRL-P Same as ↑ (see k).

CTRL-Q Not a command character. In input mode, **CTRL-Q** quotes the next character, the same as **CTRL-V**, except that some teletype drivers will eat the **CTRL-Q** so that vi never sees it. Resumes operation suspended by CTRL-S.

CTRL-R Redraws the current screen, eliminating logical lines not corresponding to physical lines (lines with only a single @ character on them). On hardcopy terminals in *open* mode, retypes the current line.

CTRL-S Some teletype drivers use **CTRL-S** to suspend output until **CTRL-Q** is pressed. Unused.

CTRL-T Not a command character. During an insert with ***autoindent*** set and at the beginning of the line, inserts ***shiftwidth*** blank space.

CTRL-U Scrolls the screen up half a window, the reverse of CTRL-D, which scrolls down. Counts work as they do for CTRL-D, and the previous scroll amount is common to both CTRL-D and CTRL-U. On a dumb terminal, CTRL-U will often necessitate clearing and redrawing the screen further back in the file.

CTRL-V Not a command character. In input mode, quotes the next character so that it is possible to insert non-printing and special characters into the file.

CTRL-W Not a command character. During an insert, backs up as b does in command mode; the deleted characters remain on the display (see CTRL-H).

CTRL-X Unused.

CTRL-Y Exposes one more line above the current screen, leaving the cursor where it is if possible. (No mnemonic value for this key; CTRL-Y is the reverse of CTRL-E).

CTRL-Z Stops the editor, exiting to the top level shell. Same as :stopCR.

CTRL-[(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 flashes the screen or rings the bell. You can thus type ESC if you don't know what is happening till the editor flashes the screen. If you don't know if you are in insert mode, you can type ESCa, and then material to be input; the material is inserted correctly whether or not you were in insert mode when you started.

CTRL-\ Unused.

CTRL-] Searches for the word which is after the cursor as a tag. Equivalent to typing :ta, this word, and then a CR. Mnemonically, this command is 'go right to'.

CTRL-^ Equivalent to :e #CR, returning to the previous position in the last-edited file, or editing a file that you specified if you got a `No write since last change diagnostic` and do not want to have to type the filename again. You have to do a :w before CTRL-^ will work in this case. If you do not wish to write the file you should do :e! #CR instead.

CTRL-_ Unused. Reserved as the command character for the Tektronix 4025 and 4027 terminal.

SPACE Same as → (see l).

! An operator that processes lines from the buffer with reformatting commands. Follow ! with the object to be processed, and then the command name terminated by CR. Doubling ! and preceding it by a count filters the count lines; otherwise the count is passed on to the object after the !. Thus 2!}fmtCR reformats the next two paragraphs by running them through the program fmt. If you are working on LISP, the command !%grindCR, given at the beginning of a function, will run the text of the function through the LISP grinder. (The grind command may not be present at all installations.) To read the output of a command into the buffer, use :r*cmd*. To simply execute a command, use :!*cmd*.

" Precedes a named buffer specification. There are named buffers 1–9 used for saving deleted text and named buffers a–z into which you can place text.

The macro character which, when followed by a number, will substitute for a function key on terminals without function keys. In input mode, if this is your erase character, it will delete the last character you typed in input mode, and must be preceded with a \ to insert it, since it normally backs over the last input character you gave.

$ Moves to the end of the current line. If you :se listCR, the end of each line is indicated by showing a $ after the end of the displayed text in the line. Given a count, advances to the count'th following end of line; thus 2$ advances to the end of the next line.

% Moves to the parenthesis or brace { } that balances the parenthesis or brace at the current cursor position.

& A synonym for :&CR, by analogy with the ex & command.

' When followed by a '', returns to the previous context at the beginning of a line. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter a–z, returns to the line that was marked with this letter with a m command, at the first non-blank character in the line. When used with an operator such as d, the operation takes place over complete lines; if you use `, the operation takes place from the exact marked place to the current cursor position within the line.

(Retreats to the beginning of a sentence, or to the beginning of a LISP s-expression if the *lisp* option is set. A sentence ends at a ., !, or ? and is followed by either the end of a line or by two spaces. Any number of closing),], ", and ' characters may appear after the ., !, or ?, and before the spaces or end of line. Sentences also begin at paragraph and section boundaries (see the "{" and "[[" entries below). A count advances that many sentences.

) Advances to the beginning of a sentence. A count repeats the effect. See (above for the definition of a sentence.

* Unused.

+ Same as CR when used as a command.

, Reverse of the last f, F, t, or T command, looking the other way in the current line. Especially useful after typing too many ; characters. A count repeats the search.

– Retreats to the previous line at the first non-blank character. This is the inverse of + and RETURN. If the line moved to is not on the screen, the screen is scrolled, or cleared and redrawn if scrolling is not possible. If a large amount of scrolling is required, the screen is also cleared and redrawn, with the current line at the center.

. Repeats the last command that changed the buffer. Especially useful when deleting words or lines; you can delete some words or lines and then type . to delete more words or lines. Given a count, it passes it on to the command being repeated. Thus after a `2dw`, `3.` deletes three words.

/ Reads a string from the last line on the screen, and scans forward for the next occurrence of this string. The normal input editing sequences may be used during the input on the bottom line; an ESC returns to command state without ever searching. The search begins when you type CR to terminate the pattern; the cursor moves to the beginning of the last line to indicate that the search is in progress; you can then terminate the search with a CTRL-C (orDEL or RUB), or by backspacing when at the beginning of the bottom line, returning the cursor to its initial position. Searches normally wrap end-around to find a string anywhere in the buffer.

When used with an operator, the enclosed region is normally affected. By mentioning an offset from the line matched by the pattern, you can affect whole lines. To do this, give a pattern with a closing / and then an offset *+n* or *−n*.

To include the character / in the search string, you must escape it with a preceding \. A ^ at the beginning of the pattern forces the match to occur at the beginning of a line only; this may speed the search. A $ at the end of the pattern forces the match to occur at the end of a line only. More extended pattern matching is available. Unless you set `nomagic` in your ***.login*** file (*?*), you will have to precede the characters ., [, *, and ~ in the search pattern with a \ to get them to work as you would naively expect.

0 Moves to the first character on the current line. Also used, in forming numbers, after an initial `1–9`.

1–9 Used to form numeric arguments to commands.

: A prefix to a set of commands for file and option manipulation and escapes to the system. Input is given on the bottom line and terminated with a CR, and the command is then executed. You can return to where you were by typing ESC or DEL if you type : accidentally.

; Repeats the last single character find that used `f`, `F`, `t`, or `T`. A count iterates the basic scan.

< An operator that shifts lines left one ***shiftwidth***, normally 8 spaces. Like all operators, affects lines when repeated, as in `<<`. Counts are passed through to the basic object, thus `3<<` shifts three lines.

= Reindents line for LISP, as though they were typed in with ***lisp*** and ***autoindent*** set.

> An operator that shifts lines right one ***shiftwidth***, normally 8 spaces. Affects lines when repeated as in `>>`. Counts repeat the basic object.

?	Scans backward, the opposite of /. See the / description above for details on scanning.
@	A macro character. If this is your kill character, you must escape it with a \ to type it in during input mode, as it normally backs over the input you have given on the current line.
A	Appends at the end of line; a synonym for $a.
B	Backs up a word, where words are composed of non-blank sequences, placing the cursor at the beginning of the word. A count repeats the effect.
C	Changes the rest of the text on the current line; a synonym for c$.
D	Deletes the rest of the text on the current line; a synonym for d$.
E	Moves forward to the end of a word, defined as blanks and non-blanks, like B and W. A count repeats the effect.
F	Finds a single following character backward in the current line. A count repeats this search that many times.
G	Goes to the line number given as preceding argument, or to the end of the file if you do not give a preceding count. The screen is redrawn with the new current line in the center if necessary.
H	Home arrow. Homes the cursor to the top line on the screen. If a count is given, the cursor is moved to the count'th line on the screen. In any case the cursor is moved to the first non-blank character on the line. If used as the target of an operator, full lines are affected.
I	Inserts at the beginning of a line; a synonym for CTRL-i.
J	Joins together lines, supplying appropriate blank space: one space between words, two spaces after a '.', and no spaces at all if the first character of the joined on line is). A count causes that many lines to be joined rather than the default two.
K	Unused.
L	Moves the cursor to the first non-blank character of the last line on the screen. With a count, to the first non-blank of the count'th line from the bottom. Operators affect whole lines when used with L.
M	Moves the cursor to the middle line on the screen, at the first non-blank position on the line.
N	Scans for the next match of the last pattern given to / or ?, but in the reverse direction; this is the reverse of n.
O	Opens a new line above the current line and inputs text there up to an ESC. A count can be used on dumb terminals to specify a number of lines to be opened; this is generally obsolete, as the ***slowopen*** option works better.
P	Puts the last deleted text back before/above the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines. Otherwise the text is inserted between the characters before and at the cursor. May be preceded by a named buffer specification "*x* to retrieve the contents of the buffer; buffers 1–9 contain deleted material, buffers a–z are available for general use.

Q — Quits from vi to ex command mode. In this mode, whole lines form commands, ending with a RETURN. You can give all the : commands; the editor supplies the : as a prompt.

R — Replaces characters on the screen with characters you type (overlay fashion). Terminates with an ESC.

S — Changes whole lines, a synonym for cc. A count substitutes for that many lines. The lines are saved in the numeric buffers, and erased on the screen before the substitution begins.

T — Takes a single following character, locates the character before the cursor in the current line, and places the cursor just after that character. A count repeats the effect. Most useful with operators such as d.

U — Restores the current line to its state before you started changing it.

V — Unused.

W — Moves forward to the beginning of a word in the current line, where words are defined as sequences of blank/non-blank characters. A count repeats the effect.

X — Deletes the character before the cursor. A count repeats the effect, but only characters on the current line are deleted.

Y — Yanks a copy of the current line into the unnamed buffer, to be put back by a later p or P; a very useful synonym for yy. A count yanks that many lines. May be preceded by a buffer name to put lines in that buffer.

ZZ — Exits the editor. (Same as :xCR.) If any changes have been made, the buffer is written out to the current file. Then the editor quits.

[[— Backs up to the previous section boundary. A section begins at each macro in the ***sections*** option, normally a .NH or .SH and also at lines that start with a formfeed CTRL-L. Lines beginning with { also stop [[; this makes it useful for looking backward, a function at a time, in C programs. If the ***lisp*** option is set, stops at each (at the beginning of a line, and is thus useful for moving backward at the top level LISP objects.

\ — Unused.

]] — Forward to a section boundary; see [[for a definition.

^ — Moves to the first non-blank position on the current line.

_ — Unused.

` — When followed by a ` returns to the previous context. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter a–z, returns to the position that was marked with this letter with an m command. When used with an operator such as d, the operation takes place from the exact marked place to the current position within the line; if you use ´, the operation takes place over complete lines.

a — Appends arbitrary text after the current cursor position; the insert can continue onto multiple lines by using RETURN within the insert. A count causes the inserted text to be replicated, but only if the inserted text is all on one line. Terminate the insertion with an ESC.

b — Backs up to the beginning of a word in the current line. A word is a sequence of alphanumerics, or a sequence of special characters. A count repeats the effect.

c — An operator that changes the following object, replacing it with the following input text up to an ESC. If more than part of a single line is affected, the text that is changed is saved in the numeric named buffers. If only part of the current line is affected, the last character to be changed away is marked with a `$`. A count causes that many objects to be affected, thus both `3c)` and `c3)` change the following three sentences.

d — An operator that deletes the following object. If more than part of a line is affected, the text is saved in the numeric buffers. A count causes that many objects to be affected; thus `3dw` is the same as `d3w`.

e — Advances to the end of the next word, defined as for `b` and `w`. A count repeats the effect.

f — Finds the first instance of the next character following the cursor on the current line. A count repeats the find.

g — Unused.

Arrow keys `h`, `j`, `k`, `l`, and CTRL-H (^H).

h — Left arrow. Moves the cursor one character to the left. Like the other arrow keys, either `h`, the left arrow key, or one of the synonyms, CTRL-H has the same effect. A count repeats the effect.

i — Inserts text before the cursor.

j — Down arrow. Moves the cursor one line down in the same column. If the position does not exist, `vi` comes as close as possible to the same column. Synonyms include CTRL-J (linefeed) and CTRL-N.

k — Up arrow. Moves the cursor up one line. CTRL-P is a synonym.

l — Right arrow. Moves the cursor one character to the right. SPACE is a synonym.

m — Marks the current position of the cursor in the mark register that is specified by the next character `a–z`. Return to this position or use with an operator using `` ` `` or `'`.

n — Repeats the last `/` or `?` scanning commands.

o — Opens new lines below the current line; otherwise like `O`.

p — Puts text after or below the cursor; otherwise like `P`.

q — Unused.

r — Replaces the single character at the cursor with a single character you type. The new character may be a RETURN; this is the easiest way to split lines. A count replaces each of the following count characters with the single character given; see `R` above which is the more usually useful iteration of `r`.

s — Changes the single character under the cursor to the text that follows up to an ESC; given a count, that many characters from the current line are changed. The last character to be changed is marked with `$` as in `c`.

t — Advances the cursor up to the character before the next character typed. Most useful with operators such as d and c to delete the characters up to a following character. You can use . to delete more if this doesn't delete enough the first time.

u — Undoes the last change made to the current buffer. If repeated, will alternate between these two states, thus is its own inverse. When used after an insert that inserted text on more than one line, the lines are saved in the numeric named buffers.

v — Unused.

w — Advances to the beginning of the next word, as defined by b.

x — Deletes the single character under the cursor. With a count deletes that many characters forward from the cursor position, but only on the current line.

y — An operator, yanks the following object into the unnamed temporary buffer. If preceded by a named buffer specification, "*x*, the text is placed in that buffer also. Text can be recovered by a later p or P.

z — Redraws the screen with the current line placed as specified by the following character: RETURN specifies the top of the screen, . the center of the screen, and '–' at the bottom of the screen. A count before the z gives the number of the line to place in the center of the screen instead of the default current line. To change the window size, use a count after the z and before the RETURN, as in z5<CR>.

{ — Retreats to 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.

CTRL-C (DEL) — Interrupts the editor, returning it to command accepting state.

2.10. Terminal Information

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

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

2.10.1. Specifying Terminal Type

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

Table 2-6: Terminal Types

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

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

```
hostname% setenv TERM 2621
```

If you are using the Bourne shell, use:

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

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

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

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

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

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

2.10.2. Special Arrangements for Startup

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

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

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

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

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

2.10.3. Open Mode on Hardcopy Terminals and 'Glass tty's'

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

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

- z and
- CTRL-R.

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

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

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

2.10.4. Editing on Slow Terminals

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ignoring the current state of the display.

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

2.10.5. Upper-case Only Terminals

If your terminal has only upper-case characters, you can still use vi by using the normal system convention for typing on such a terminal. Characters that you normally type are converted to lower case, and you can type upper-case letters by preceding them with a '\'. The characters { ~ } | ` are not available on such terminals, but you can escape them as \(\^ \) \! \'. These characters are represented on the display in the same way they are typed.[11]

2.11. Command Summary

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

[11] The '\' character you give will not echo until you type another key.

Table 2-7: Frequently-Used vi Commands

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

^	move cursor to first non-blank on line
$	move cursor to end of line
)	move cursor forward one sentence
(	move cursor backward one sentence
}	move cursor forward one paragraph
{	move cursor backward one paragraph
]]	move cursor forward one section
[[	move cursor backward one section
F*x*	find *x* backward in line
f*x*	find *x* forward in line
P	put text back, before cursor or above current line
p	put text back, after cursor or below current line
Y	yank one line into buffer
y	yank the object you specify into buffer; for copies and moves
t*x*	perform some operation forward on the line to *x*
T*x*	perform some operation backward on the line to *x*

Vi Quick Reference

Entering/Leaving vi

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

The Display

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

Vi Modes

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

Counts Before vi Commands

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

Simple Commands

dw	delete a word
de	... leaving punctuation
dd	delete a line
3dd	... 3 lines
i*text*ESC	insert text *abc*
cw*new*ESC	change word to *new*
ea*s*ESC	pluralize word
xp	transpose characters

Interrupting, Cancelling

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

File Manipulation

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

Positioning within File

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

Adjusting the Screen

^L	clear and redraw
^R	retype, eliminate @ lines
z CR	redraw, current at window top
z-	... at bottom
z.	... at center
/*pat*/z-	*pat* line at bottom
z*n*.	use *n* line window
^E	scroll window down 1 line
^Y	scroll window up 1 line

Marking and Returning

``	previous context
''	... at first non-white in line
m*x*	mark position with letter *x*
`*x*	to mark *x*
'*x*	... at first non-white in line

Line Positioning

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

Character Positioning

^	first non white
0	beginning of line
$	end of line
h or →	forward
l or ←	backwards
^H	same as ←
space	same as →
f*x*	find *x* forward
F*x*	f backward
t*x*	upto *x* forward
T*x*	back upto *x*
;	repeat last f F t or T
,	inverse of ;
|	to specified column
%	find matching ({) or }

Words, Sentences, Paragraphs

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

Commands for LISP

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

Corrections During Insert

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

Insert and Replace

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

Operators (double to affect lines)

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

Miscellaneous Operations

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

Yank and Put

p	put back lines
P	put before
"*x*p	put from buffer *x*
"*x*y	yank to buffer *x*
"*x*d	delete into buffer *x*

Undo, Redo, Retrieve

u	undo last change
U	restore current line
.	repeat last change
"*d*p	retrieve *d*'th last delete

Chapter 3

Command Reference for the ex Line Editor

This chapter[12] provides reference material for **ex**, the line-oriented text editor, which also supports display oriented editing in the form of the **vi** editor described in the chapter "Using **vi**, the Visual Display Editor". The contents of this chapter describe the line-oriented part of **ex**. You can also use these commands with **vi**. For a summary of **ex** commands, see the **ex** Quick Reference.

3.1. Using ex

ex has a set of options, which you can use to tailor **ex** to your liking. The command **edit** invokes a version of **ex** designed for more casual or beginning users by changing the default settings of some of these options. To simplify the description which follows, we assume the default settings of the options, and we assume that you are running **ex** on a Sun Workstation.

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

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

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

hostname% **ex** [**−**] [**−v**] [**−t** ***tag***] [**−r**] [**−l**] [**−w*n***] [**−x**] [**−R**] [**+*command***] ***filename*** ...

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

hostname% **ex** ***filename***

The '−' command line option option suppresses all interactive-user feedback and is useful in processing **ex** scripts in command files. The **−v** option is equivalent to using **vi** rather than **ex**. The **−t** option is equivalent to an initial ***tag*** command, editing the file containing the ***tag*** and positioning the editor at its definition.

Use the **−r** option to recover a file after an editor or system problem, retrieving the last saved version of the named file or, if no file is specified, displaying a list of saved files. The **−l** option sets up for editing LISP, setting the ***showmatch*** and ***lisp*** options. The **−w** option sets the default window size to ***n***, and is useful on dialups to start in small windows. The **−x** option causes **ex**

[12] The material in this chapter is derived from *Ex Reference Manual*, W.N. Joy, M. Horton, University of California, Berkeley.

to prompt for a ***key***, which is used to encrypt and decrypt the contents of the file, which should already be encrypted using the same key (see ***crypt*** in the ***Commands Reference Manual for the Sun Workstation***). The -R option sets the ***readonly*** option at the start. If set, writes will fail unless you use an ! after the write. This option affects ZZ, ***autowrite*** and anything that writes to guarantee you won't clobber a file by accident. ***Filename*** arguments indicate files to be edited. An argument of the form +***command*** indicates that the editor should begin by executing the specified command. If ***command*** is omitted, it defaults to '$', initially positioning ex at the last line of the first file. Other useful commands here are scanning patterns of the form '***/pat***' or line numbers, such as +100, which means 'start at line 100.'

3.2. File Manipulation

The following describes commands for handling files.

3.2.1. Current File

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

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

3.2.2. Alternate File

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

3.2.3. Filename Expansion

You may specify filenames within the editor using the normal Shell expansion conventions. In addition, the character % in filenames is replaced by the ***current*** filename and the character # by the ***alternate*** filename. This makes it easy to deal alternately with two files and eliminates the need for retyping the name supplied on an `edit` command after a `No write since last change` diagnostic is received.

3.3. Special Characters

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

```
.    &    *    [    ]    ~
```

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

3.3.1. Multiple Files and Named Buffers

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

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

3.3.2. Read Only Mode

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

3.4. Exceptional Conditions

The following describes additional editing situations.

3.4.1. Errors and Interrupts

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

3.4.2. Recovering If Something Goes Wrong

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

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

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

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

```
hostname% ex -r
```

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

3.5. Editing Modes

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

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

3.6. Command Structure

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

3.6.1. Specifying Command Parameters

Most commands accept prefix addresses specifying the lines in the file upon which they are to have effect. The forms of these addresses will be discussed below. A number of commands also may take a trailing ***count*** specifying the number of lines to be involved in the command. Counts are rounded down if necessary. Thus the command **10p** displays the tenth line in the buffer,

while `d5` deletes five lines from the buffer, starting with the current line.

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

3.6.2. Invoking Command Variants

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

3.6.3. Flags after Commands

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

3.6.4. Writing Comments

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

3.6.5. Putting Multiple Commands on a Line

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

3.6.6. Reporting Large Changes

Most commands which change the contents of the editor buffer give feedback if the scope of the change exceeds a threshold given by the ***report*** option. This feedback helps to detect undesirably large changes so that you may quickly and easily reverse them with ***undo***. After commands with more global effect, such as ***global*** or ***visual,*** you will be informed if the net change in the number of lines in the buffer during this command exceeds this threshold.

3.7. Command Addressing

The following describes the editor commands called ***addressing primitives.***

3.7.1. Addressing Primitives

`.` The current line. The current line is traditionally called 'dot' because you address it with a dot '`.`'. Most commands leave the current line as the last line which they affect. The default address for most commands is the current line, so you rarely use '`.`' alone as an address.

n The ***n***th line in the editor's buffer, lines being numbered sequentially from 1.

`$` The last line in the buffer.

`%` An abbreviation for `1,$`, the entire buffer.

`+`***n*** `-`***n*** An offset relative to the current buffer line. The forms `.+3` `+3` and `+++` are all equivalent; if the current line is line 100, they all address line 103.

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

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

3.7.2. 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 you give more addresses than the command requires, all but the last one or two are ignored. If the command takes two addresses, the first addressed line must precede the second in the buffer. Null address specifications are permitted in a list of addresses; the default in this case is the current line '`.`'. So ',100' is equivalent to '`.`,100'. It is an error to give a prefix address to a command which expects none.

3.8. Regular Expressions and Substitute Replacement Patterns

3.8.1. 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 regular expression, that is **//** or **??**.

3.8.2. 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 backslash (\) to use them as "ordinary" characters. With ***nomagic***, the default for **edit**, regular expressions are much simpler because there are only two metacharacters: '^' (beginning of line) and '$' (end of line). The power of the other metacharacters is still available by preceding the (now) ordinary character with a \. Note that \ is thus always a metacharacter.

The remainder of the discussion of regular expressions assumes that the setting of this option is ***magic***[13]

3.8.3. Basic 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,' that is either the end of the line or before character which is neither a letter, nor a digit, nor the underline character.

---

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

[*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 a set of characters 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 a `^`, the construct matches all but those characters; thus `[^a–z]` matches anything but a lower-case letter and of course a newline. You must escape any of the characters `^`, `[`, or – in ***string*** with a preceding `\`.

3.8.4. 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 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.

3.8.5. 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 escape character `\`. The sequence '`\`*n*' is replaced by the text matched by the *n*-th regular subexpression enclosed between `\(` and `\)`.[14] The sequences `\u` and `\l` cause the immediately following character in the replacement to be converted to upper- or lower-case respectively if this character is a letter. The sequences `\U` and `\L` turn such conversion on, either until `\E` or `\e` is encountered, or until the end of the replacement pattern.

3.9. Command Reference

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

address command ! parameters count flags

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

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

[14] When nested, parenthesized subexpressions are present, *n* is determined by counting occurrences of `\(` starting from the left.

abbreviate ***word rhs*** abbr: **ab**

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

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

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

a!
text
.

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

args

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

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

Replaces the specified lines with the input ***text***. The current line becomes the last line input; if no lines were input, it is left as for a **delete**.

c!
text
.

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

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

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

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

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

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

Used to begin an editing session on a new file. Same as **:vi** ***file***. The editor first checks to see if the buffer has been modified since the last **write** command was issued. If it has

been, a warning is issued and the command is aborted. The command otherwise deletes the entire contents of the editor buffer, makes the named file the current file and prints the new filename. After insuring that this file is sensible the editor reads the file into its buffer. A 'sensible' file is not a binary file such as a directory, a block or character special file other than */dev/tty*, a terminal, or a binary or executable file as indicated by the first word.

If the read of the file completes without error, the number of lines and characters read is typed. If there were any non-ASCII characters in the file they are stripped of their non-ASCII high bits, and any null characters in the file are discarded. If none of these errors occurred, the file is considered ***edited.*** If the last line of the input file is missing the trailing newline character, it will be supplied and a complaint will be issued. This command leaves the 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 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, for example: *+/pat.*

`file` abbr: `f`

Prints 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 also noted. You have to use `w!` to write to the file, since `ex` does not want to `write` a file unrelated to the current contents of the buffer.

`file` *file*

The current filename is changed to *file* which is considered `[Not edited]`.

`( 1 , $ ) global` */pat/ cmds* abbr: `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 possibly the trailing / delimiter) 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 and is not changed during a global command, except perhaps by an

open or visual command within the global.

g! */pat/ cmds* abbr: v

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

(.)insert abbr: i
text
.

Places the given text before the specified line. The current line is left at the last line input; if there were none input it is left at the line before the addressed line. This command differs from append only in the placement of text.

i!
text
.

The variant toggles *autoindent* during the insert.

(. , .+1) join *count flags* abbr: j

Places the text from a specified range of lines together on one line. White space is adjusted at each junction to provide at least one blank character, two if there was a '.' at the end of the line, or none if the first following character is a). If there is already white space at the end of the line, then the white space at the start of the next line will be discarded.

j!

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

(.) k *x*

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

(. , .) list *count flags*

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

map *lhs rhs*

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

(.) mark *x*

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

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

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

next abbr: n

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

n!

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

n *filelist*
n +*command filelist*

The specified *filelist* is expanded and the resulting list replaces the current argument list; the first file in the new list is then edited. If *command* is given (it must contain no spaces), then it is executed after editing the first such file.

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

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

(.) open *flags* abbr: o
(.) open */pat/ flags*

Enters intraline editing *open* mode at each addressed line. If *pat* is given, then the cursor will be placed initially at the beginning of the string matched by the pattern. To exit this mode, use Q. See the chapter on "Using vi the Visual Display Editor".

preserve

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

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

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

(.)put *buffer* abbr: pu

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

quit abbr: **q**

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

q!

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

(.) **read** *file* abbr: **r**

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

Address '0' (zero) is legal for this command and causes the file to be read at the beginning of the buffer. Statistics are given as for the **edit** command when the ***read*** successfully terminates. After a **read** the current line is the last line read. Within ***open*** and ***visual*** modes the current line is set to the first line read rather than the last.

(.) **read** **!*command***

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

recover *file*

Recovers *file* from the system save area. Used after an accidental hangup of the phone or a system crash or **preserve** command. The system saves a copy of the file you were editing only if you have made changes to the file. Except when you use **preserve** you will be notified by mail when a file is saved.

rewind abbr: **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 no*option*** 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`; they are interpreted from left to right.

`shell` abbr: `sh`

A new shell is created. When it terminates, editing resumes.

`source` *file* abbr: `so`

Reads and executes commands from the specified file. `source` commands may be nested.

(. , .) `substitute` */pat/repl/ options count flags* abbr: `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` one can cause the substitution to be performed, any other input causes no change to take place. After a `substitute` command is executed, the last line substituted becomes the current line.

Lines may be split by substituting new-line 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. This commands is only available where supported by the teletype driver and operating system.

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

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

(. , .) `t` *addr flags*

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

`ta` *tag*

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

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

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

`unabbreviate` ***word*** abbr: `una`

Delete ***word*** from the list of abbreviations.

`undo` abbr: `u`

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

`undo` always marks the previous value of the current line '.' as ''''. After an `undo` the current line is the first line restored or the line before the first line deleted if no lines were restored. For commands with more global effect such as `global` and `visual` the current line regains its pre-command value after an `undo`.

`unmap` ***lhs***

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

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

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

`version` abbr: `ve`

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

`vi` ***file***

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

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

Enters visual mode at the specified line. ***Type*** is optional and may be '–' , '^' or '.' as in the `z` command to specify the placement of the specified line on the screen. By default, if ***type*** is omitted, the specified line is placed as the first on the screen. A ***count*** specifies an initial window size; the default is the value of the option ***window.*** See the chapter "Using `vi`, the Visual Display Editor for more details. To exit visual mode, type `Q`.

`visual` file
`visual +`***n*** `file`

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

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

Writes changes made back to ***file***, printing the number of lines and characters written. Normally ***file*** is omitted and the text goes back where it came from. If a ***file*** is specified, then text will be written to that file.[15] If the file does not exist it is created. The current file

[15] 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.

name is changed only if there is no current file name; the current line is never changed.

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

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

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

w! *name*

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

(1 , $) w !*command*

Writes the specified lines into ***command.*** Note the difference between w! which overrides checks and w ! which writes to a command.

wq *name*

Like a write and then a quit command.

wq! *name*

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

xit *name* abbr: x

If any changes have been made and not written, writes the buffer out. Then, in any case, quits. Same as wq, but does not bother to write if there have not been any changes to the file.

(. , .)yank ***buffer count*** abbr: 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 ***window.***

(.) z *type count*

Displays a window of text with the specified line at the top. If ***type*** is '—' the line is placed at the bottom; a '.' places the line 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 terminal, the screen is cleared before display begins unless you give a ***count*** less than the screen size. The current line is left at the last line displayed. 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 resulting output then replaces the input lines.

($) =

Prints the line number of the addressed line. The current line is unchanged.

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

Perform intelligent shifting on the specified lines; < shifts left and > shifts right. The quantity of shift is determined by the *shiftwidth* option and the repetition of the specification character. Only white space (blanks and tabs) is shifted; no non-white characters are discarded in a left-shift. The current line becomes the last line which changed due to the shifting.

CTRL-D

An end-of-file from a terminal input scrolls through the file. The *scroll* option specifies the size of the scroll, normally a half screen of text.

(.+1 , .+1)
(.+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.

3.10. Option Descriptions

autoindent, ai default: noai

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

If you then type in 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 be aligned with the first non-white character of the previous line. To back the cursor up to the preceding tab stop, type CTRL-D. The tab stops going backwards are defined at multiples of the ***shiftwidth*** option. You ***cannot*** backspace over the indent, except by sending an end-of-file with a CTRL-D.

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

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

autoprint, ap default: ap

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, aw default: noaw

Causes the contents of the buffer to be written to the current file if you have modified it and give a `next`, `rewind`, `stop`, `tag`, or `!` command, or a CTRL-^ (switch files) or CTRL-] (tag goto) command in ***visual*** mode. Note, that 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*** (`edit` for `next`, `rewind!` for `rewind`, `stop!` for `stop`, `tag!` for `tag`, `shell` for `!`, and `:e #` and a `:ta!` command from within ***visual*** mode).

beautify, bf default: nobeautify

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

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

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

edcompatible default: noedcompatible

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

errorbells, eb default: noeb

Error messages are preceded by a beep or bell.[16] If possible the editor always places the error message in a standout mode of the terminal (such as inverse video) instead of ringing the bell.

hardtabs, ht default: ht=8

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

ignorecase, ic default: noic

All upper case characters in the text are mapped to lower case in regular expression matching. In addition, all upper case characters in regular expressions are mapped to lower case except in character class specifications.

lisp default: nolisp

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

list default: nolist

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

magic default: magic for **ex** and `vi`[17]

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 preceding them with a backslash (\).

mesg default: mesg

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

number, nu default: nonumber

Causes all output lines to be printed with their line numbers. In addition each input line will be prompted for by supplying the line number it will have.

---

[16] Beeping and bell ringing in *open* and *visual* on errors is not suppressed by setting *noeb*.

[17] *nomagic* for **edit**.

open default: open

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

optimize, opt default: optimize

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

paragraphs, para default: para=IPLPPPQPP LIbp

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

prompt default: prompt

Command mode input is prompted for with a :.

readonly, ro default: off

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

redraw default: noredraw

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

remap default: remap

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

report default: report=5[18]

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

scroll default: scroll=½ 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*).

[18] 2 for `edit`.

sections default: sections=SHNHH HU

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

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

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

shiftwidth, sw default: sw=8

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

showmatch, sm default: nosm

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

slowopen, slow terminal dependent

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 the chapter "Using `vi`, the Visual Display Editor" for more details.

tabstop, ts default: ts=8

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

taglength, tl default: tl=0

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

tags default: tags=tags ***/usr/lib/tags***

A path of files to be used as tag files for the `tag` command, similar to the ***path*** variable of ***csh***. Separate the files by spaces, and precede each space with a backslash. Files are searched left to right. Always put ***tags*** as your first entry. A requested tag is searched for in the specified files, sequentially. By default (even in version 2) files called ***tags*** are searched for in the current directory and in ***/usr/lib*** (a master file for the entire system.)

term default: from environment TERM

The terminal type of the output device.

terse default: noterse

Shorter error diagnostics are produced for the experienced user.

timeout default: on

Causes macros to time out after one second. Turn it off and they wait forever. Use this if you want multi-character macros. If your terminal sends an escape sequence for arrow keys, type ESC twice.

warn default: warn

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

window default: window=speed dependent

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.

w300, w1200, w9600

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

wrapscan, ws default: ws

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

wrapmargin, wm default: wm=0

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

writeany, wa default: nowa

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

3.11. Limitations

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

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

Ex Quick Reference

Entering/Leaving ex

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

ex States

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

ex Commands

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

ex Command Addresses

n	line *n*	*/pat*	next with *pat*
.	current	?*pat*	previous with *pat*
$	last	*x-n*	*n* before *x*
+	next	*x,y*	*x* through *y*
–	previous	*'x*	marked with *x*
+*n*	*n* forward	''	previous context
%	1,$		

Specifying Terminal Type

% setenv TERM *type*	(for *csh*)
$ TERM=*type*; export TERM	(for *sh*)

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

Some Terminal Types

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

Initialising Options

EXINIT	place set's here in environment var.
set *x*	enable option
set no*x*	disable option
set *x*=*val*	give value *val*
set	show changed options
set all	show all options
set *x*?	show value of option *x*

Useful Options

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

Scanning Pattern Formation

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

Chapter 4

Using the ed Line Editor

This chapter[19] describes the editing tools of the **ed** line editor. It provides the newcomer with elementary instructions and exercises for learning the most necessary and common commands and the more advanced user with information about additional editing facilities. The contents include descriptions of appending, changing, deleting, moving, copying and inserting lines of text; reading and writing files; displaying your files; context searching; the global commands; line addressing; and using special characters. There are also brief discussions on writing scripts and on the pattern-matching tool **grep**, which is related to **ed**.

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

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

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

4.1. Getting Started

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

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

```
hostname%
```

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

```
hostname% ed
hostname%
```

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

---

[19] The material in this chapter is derived from *A Tutorial Introduction to the UNIX Text Editor*, B.W. Kernighan and *Advanced Editing on UNIX*, B.W. Kernighan, Bell Laboratories, Murray Hill, New Jersey.

4.1.1. Creating Text — the Append Command a

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

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

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

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

After the append command, your file contains the lines:

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

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

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

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

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

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

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

4.1.2. Error Messages — ?

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

```
?
```

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

4.1.3. Writing Text Out as a File — the Write Command `w`

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

```
w junk
68
```

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

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

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

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

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

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

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

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

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

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

4.1.4. Leaving ed *— the Quit Command* q

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

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

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

4.1.5. Creating a New File — the Edit Command e

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

```
e  newfile
```

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

If you enter ed with the command:

```
hostname% ed  file
```

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

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

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

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

```
hostname% e junk
68
```

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

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

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

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

4.1.6. Exercise: Trying the e *Command*

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

```
?name
```

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

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

is equivalent to:

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

4.1.7. Checking the Filename — the Filename Command f

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

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

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

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

4.1.8. Reading Text from a File — the Read Command r

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

```
r junk
68
```

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

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

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

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

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

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

Table 1 shows that ...

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

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

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

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

4.1.9. Printing the Buffer Contents — the Print Command `p`

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

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

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

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

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

```
CTRL-C
?
```

`ed` waits for the next command.

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

```
$,$p
to come to the aid of their party.
```

or abbreviate it to:

```
$p
to come to the aid of their party.
```

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

```
1p
Now is the time
```

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

```
2
for all good men
```

`ed` displays the second line of the buffer.

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

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

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

4.1.10. Exercise: Trying the `p` *Command*

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

```
3,1p
?
```

4.1.11. Displaying Text — the List Command `l`

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

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

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

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

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

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

4.1.12. The Current Line — 'Dot' or '.'

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

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

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

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

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

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

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

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

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

In our example these are lines 3 through 6.

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

Dot is most useful in combinations like:

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

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

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

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

```
.-3,.-1p
```

This command displays the previous three lines.

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

```
.=
3
```

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

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

4.1.13. Deleting Lines — the Delete Command d

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

starting line, ending line **d**

Thus the command:

```
4,$d
```

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

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

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

4.1.14. Exercise: Experimenting

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

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

```
0r filename
number of characters
```

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

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

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

```
.
w
number of characters
```

4.1.15. Modifying Text — the Substitute Command s

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

Suppose that by a typing error, line 1 says:

```
Now is th time
```

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

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

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

```
p
Now is the time
```

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

The general way to use the substitute command is:

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

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in ***all*** the lines between ***starting-line*** and ***ending-line***. Only the first occurrence on each line is changed, however. If you want to change ***every*** occurrence, read on below. The rules for line numbers are the same as those for p, except that dot is set to the last line changed. But there is a trap for the unwary: if no substitution took place, dot is ***not*** changed. This causes an error '?' as a warning.

Thus you can say:

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

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

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

every line of the file, type:

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

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

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

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

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

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

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

If it didn't, you can try again.

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

You can also say:

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

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

```
Nowxx is the time
```

To correct this, say:

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

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

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

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

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

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

The general format is:

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

Try other characters instead of slashes to delimit the two sets of characters in the **s** command — anything should work except blanks or tabs.

If you get funny results using any of the characters:

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

read the section on "Special Characters".

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

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

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

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

4.1.16. The Ampersand &

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

```
Now is the time
```

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

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

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

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

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

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

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

or

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

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

```
the end of the world
```

you can type:

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

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

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

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

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

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

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

4.1.17. Exercise: Trying the s *and* g *Commands*

Experiment with s and g. See what happens if you substitute for some word on a line with several occurrences of that word. For example, do this:

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

4.1.18. Undoing a Command — the Undo Command u

Occasionally you will make a substitution in a line, only to realize too late that it was a mistake. Use the ***undo*** command u to undo the last substitution. This restores the last line that was substituted to its previous state. For example, study the following example:

```
s/party/country/
p
to come to the aid of their country.
u
p
to come to the aid of their party.
```

4.2. Changing and Inserting Text — the c and i Commands

This section discusses the ***change*** command c and the ***insert*** command i. The change command changes or replaces a group of one or more lines. The insert command inserts a group of one or more lines.

The c command replaces a number of lines with different lines you type in at the workstation. For example, to change lines '.+1' through '$' to something else, type:

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

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

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

'Insert' is similar to append, for instance:

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

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

4.2.1. Exercise: Trying the `c` Command

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

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

is almost the same as:

```
start, end c
. . . text . . .
.
```

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

Experiment with `a` and `i`, to see that they are similar, but not the same. You will observe that to append ***after*** the given line, you type:

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

while to insert ***before*** it, you type:

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

Observe that if you do not give a line number, `i` inserts before line dot, while `a` appends after line dot.

4.3. Specifying Lines in the Editor

To specify which lines are to be affected by the editing commands, you use ***line addressing.*** There are several methods, and they are described below.

4.3.1. Context Searching

One way is ***context searching.*** Context searching is simply a method of specifying the desired line, regardless of what its number is, by specifying some context on it.

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

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

If you want to find the line that contains 'their' so you can change it to 'the'. With only three lines in the buffer, it's pretty easy to keep track of what line the word 'their' is on. But if the buffer contains several hundred lines, and you'd been making changes, deleting and rearranging lines, and so on, you would no longer really know what this line number would be.

For example, to locate the next occurrence of the characters between slashes ('their'), type:

```
/their/
to come to the aid of their party.
```

To search for a line that contains a particular string of characters, the general format is:

/string of characters we want to find/

This is sufficient to find the desired line. It also sets dot to that line and displays the line for verification. 'Next occurrence' means that ed starts looking for the string at line '.+1', searches to the end of the buffer, then continues at line 1 and searches to line dot. That is, the search 'wraps around' from '$' to 1. It scans all the lines in the buffer until it either finds the desired line or gets back to dot again. If the given string of characters can't be found in any line, ed displays the error message:

```
?
```

Otherwise it shows the line it found.

Less familiar is the use of:

```
?thing?
```

This command scans ***backward*** for the previous occurrence of 'thing'. This is especially handy when you realize that the thing you want to operate on is back up the page from where you are currently editing.

The slash and question mark are the only characters you can use to delimit a context search, though you can use essentially any character in a substitute command. You can do both the search for the desired line ***and*** a substitution all at once, like this:

```
/their/s/their/the/p
to come to the aid of the party.
```

There were three parts to that last command: a context search for the desired line, the substitution, and displaying the line.

The expression `/their/` is a context search expression. In their simplest form, all context search expressions are like this — a string of characters surrounded by slashes. Context searches are interchangeable with line numbers, so you can use them by themselves to find and show a desired line, or as line numbers for some other command, like `s`. We use them both ways in the examples above.

4.3.2. Exercise: Trying Context Searching

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

Try using context searches as line numbers for the substitute, print and delete commands. You can also use context searching with with `r`, `w`, and `a`.

If you get funny results with any of the characters:

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

read the section on "Special Characters".

4.3.3. Specifying Lines with Address Arithmetic — + and −

Another area where you can save typing in specifying lines is to use minus (-) and plus (+) as line numbers by themselves. To move back up one line in the file, type:

```
-
```

In fact, you can string several minus signs together to move back up that many lines:

```
---
```

moves up three lines, as does `-3`. Thus:

```
-3,+3p
```

is also identical to the examples above.

Since `-` is shorter than `.-1`, use it to change 'bad' to 'good' on the previous line and on the current line.

```
-,.s/bad/good/
```

You can use `+` and `-` in combination with searches using `/.../` and `?...?`, and with `$`. To find the line containing 'thing', and position you two lines before it, type:

```
/thing/--
```

The next step is to combine the line numbers like '.' and '$', context searches like '/.../' and '?...?' with '+' and '−'. Thus:

```
$-1
```

displays the next-to-last line of the current file, that is, one line before line '$'. For example, to recall how far you got in a previous editing session, type:

```
$-5,$p
```

which shows the last six lines. (Be sure you understand why it shows six, not five.) If there are less than six, of course, you'll get an error message. Suppose the buffer contains the three

familiar lines:

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

Then the ed line numbers:

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

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

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

or:

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

or:

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

Convenience dictates the choice. You could display all three lines by, for instance:

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

or:

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

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

```
1,$p
```

but not if there were several hundred.

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

As another example:

```
.-3,.+3p
```

displays from three lines before where you are now at line dot to three lines after, thus giving you a bit of context. By the way, you can omit the '+':

```
.-3,.3p
```

is identical in meaning.

4.3.4. Repeated Searches — // and ??

Suppose you ask for the search:

```
/horrible thing/
```

and when the line is displayed, you discover that it isn't the horrible thing that you wanted, so you have to repeat the search again. You don't have to re-type the search; use the construction:

```
//
```

as a shorthand for 'the previous thing that was searched for', whatever it was. You can repeat this as many times as necessary. You can also search backward through the file by typing:

```
??
```

?? searches for the same thing, but in the reverse direction.

Not only can you repeat the search, but you can use '//' as the left side of a substitute command, to mean 'the most recent pattern.'

```
/horrible thing/
  ... ed prints line with 'horrible thing' ...
s//good/p
```

To go backward and change a line, say:

```
??s//good/
```

You can still use the & on the right hand side of a substitute to stand for whatever got matched:

```
//s//& &/p
```

This finds the next occurrence of whatever you searched for last, replaces it by two copies of itself, then displays the line just to verify that it worked.

4.3.5. Default Line Numbers and the Value of Dot

One of the most effective ways to speed up your editing is always to know what lines will be affected by a command if you don't specify the lines it is to act on, and on what line you will be positioned, that is, the value of dot, when a command finishes. If you can edit without specifying unnecessary line numbers, you can save a lot of typing.

As the most obvious example, if you give a search command like:

```
/thing/
```

you are left pointing at the next line that contains 'thing'. No address is required with commands like s to make a substitution on that line. Addresses are also not required with p to show it, l to list it, d to delete it, a to append text after it, c to change it, or i to insert text before it.

What would happen if there were no 'thing'? Then you are left right where you were — dot is unchanged. This is also true if you are sitting on the only 'thing' when you issue the command. The same rules hold for searches that use ?...?; the only difference is the direction in which you search.

The delete command d leaves dot pointing at the line that followed the last deleted line. When line '$' gets deleted, however, dot points at the ***new*** line '$'.

The line-changing commands a, c and i by default all affect the current line. If you do not give a line number with them, the a appends text after the current line, c changes the current line, and i inserts text before the current line.

The a, c, and i commands behave identically in one respect — when you stop appending, changing or inserting, dot points at the last line entered. This is exactly what you want for typing and editing on the fly. For example, you can say:

```
a
... text ...
... botch ...          (minor error)
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 do not add ***any*** lines with `a`, `c` or `i`.

The `r` command reads a file into the text being edited, either at the end if you do not give an address, or after the specified line if you do. In either case, dot points at the last line read in. Remember that you can even say `0r` to read a file in at the beginning of the text. You can also say `0a` or `1i` to start adding text at the beginning.

The `w` command writes out the entire file. If you precede the command by one line number, that line is written, while if you precede it by two line numbers, that range of lines is written. The `w` command does ***not*** change dot; the current line remains the same, regardless of what lines are written. This is true even if you say something that involves a context search, such as:

```
/^\.AB/,/^\.AE/w abstract
```

Since `w` is so easy to use, you should save what you are editing regularly as you go along just in case something goes wrong, or in case you do something foolish, like clobbering what you're editing.

With the `s` command, the rule is simple; you are left positioned on the last line that got changed. If there were no changes, dot doesn't move.

To illustrate, suppose that there are three lines in the buffer, and the cursor is sitting on the middle one:

```
x1
x2
x3
```

The command line

```
-,+s/x/y/p
```

displays the third line, the last one changed. But if the three lines had been:

```
x1
y2
y3
```

and the same command had been issued while dot pointed at the second line, then the result would be to change and show only the first line, and that is where dot would be set.

4.3.6. Combining Commands — the Semicolon ;

Searches with /.../ and ?...? start at the current line and move forward or backward respectively until they either find the pattern or get back to the current line. Sometimes this is not what is wanted. Suppose, for example, that the buffer contains lines like this:

```
.
.
.
ab
.
.
.
bc
.
.
```

Starting at line 1, one would expect that the command:

```
/a/,/b/p
```

would display all the lines from the 'ab' to the 'bc' inclusive. Actually this is not what happens. *Both* searches (for 'a' and for 'b') start from the same point, and thus they both find the line that contains 'ab'. The result is to display a single line. Worse, if there had been a line with a 'b' in it before the 'ab' line, then the print command would be in error, since the second line number would be less than the first, and you cannot display lines in reverse order.

This happens because the comma separator for line numbers doesn't set dot as each address is processed; each search starts from the same place. In **ed**, you can use the semicolon ; just like comma, with the single difference that use of a semicolon forces dot to be set at that point as the line numbers are being evaluated. In effect, the semicolon 'moves' dot. Thus in the example above, the command:

```
/a/;/b/p
```

displays the range of lines from 'ab' to 'bc', because after the 'a' is found, dot is set to that line, and then 'b' is searched for, starting beyond that line.

Use the semicolon when you want to find the ***second*** occurrence of something. For example, to find the second occurrence of 'thing', you can say:

```
/thing/
line with 'thing'
//
second line with 'thing'
```

But this displays the first occurrence as well as the second, and is a nuisance when you know very well that it is only the second one you're interested in. The solution is to find the first occurrence of 'thing', set dot to that line, then find the second and display only that:

```
/thing/;//
```

Closely related is searching for the second previous occurrence of something, as in:

```
?something?;??
```

We leave you to try showing the third or fourth or ... in either direction.

Finally, bear in mind that if you want to find the first occurrence of something in a file, starting at an arbitrary place within the file, it is not sufficient to say:

```
1;/thing/
```

This search fails if 'thing' occurs on line 1. But it is possible to say:

```
0;/thing/
```

This is one of the few places where 0 is a legal line number, for this starts the search at line 1.

4.3.7. Interrupting the Editor

As a final note on what dot gets set to, be aware that if you type an INTERRUPT (CTRL-C is the default, but your terminal may be set up with the DELETE, RUBOUT or BREAK keys) while ed is doing a command, things are put back together again and your state is restored as much as possible to what it was before the command began. Naturally, some changes are irrevocable — if you are reading or writing a file or making substitutions or deleting lines, these will be stopped in the middle of execution in some clean but unpredictable state; hence it is not usually wise to stop them. Dot may or may not be changed.

Displaying is more clear cut. Dot is not changed until the display is done. Thus if you display lines until you see an interesting one, then type CTRL-C, you are ***not*** sitting on that line or even near it. Dot is left where it was when the p command was started.

4.4. Editing All Lines — the Global Commands g and v

Use the *global* command g to execute one or more ed commands on all those lines in the buffer that match some specified string. For example, to display all lines that contain 'peling', type:

```
g/peling/p
```

As another example:

```
g/^\./p
```

displays all the formatting commands in a file. The pattern that goes between the slashes can be anything that could be used in a line search or in a substitute command; the same rules and limitations apply.

For a more useful command, which makes the substitution everywhere on the line, then displays each corrected line, type:

```
g/peling/s//pelling/gp
```

Compare this to the following command line, which only displays the last line substituted:

```
1,$s/peling/pelling/gp
```

Another subtle difference is that the g command does not give a '?' if 'peling' is not found whereas the s command will.

The substitute command is probably the most useful command that can follow a global because you can use this to make a change and display each affected line for verification. For example, you can change the word 'SUN' to 'Sun' everywhere in a file, and verify that it really worked, with:

```
g/SUN/s//Sun/gp
```

Notice that you use // in the substitute command to mean 'the previous pattern', in this case, 'SUN'. The p command is done on every line that matches the pattern, not just those on which a substitution took place.

The v command is identical to g, except that it operates on those lines that do *not* contain an occurrence of the pattern; that is, v 'inverts' the process, so:

```
v/^\./p
```

The command that follows g or v can be anything:

```
g/^\./d
```

deletes all lines that begin with '.', and:

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

deletes all empty lines.

The global command operates by making two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each marked line in turn is examined, dot is set to that line, and the command executed. This means that it is possible for the command that follows a g or v to use addresses, set dot, and so on, quite freely.

```
g/^\.PP/+
```

displays the line that follows each .PP command (the signal for a new paragraph in some formatting packages). Remember that + means 'one line past dot'. And:

```
g/topic/?^\.SH?1
```

searches for each line that contains 'topic', scans backward until it finds a line that begins .SH (a section heading) and shows the line that follows that, thus showing the section headings under which 'topic' is mentioned. Finally:

```
g/^\.EQ/+,/^\.EN/-p
```

displays all the lines that lie between lines beginning with .EQ and .EN formatting commands.

You can also precede the g and v commands by line numbers, in which case the lines searched are only those in the range specified.

4.4.1. Multi-line Global Commands

You can use more than one command under the control of a global command, although the syntax for expressing the operation is not especially natural or pleasant. As an example, suppose the task is to change 'x' to 'y' and 'a' to 'b' on all lines that contain 'thing'. Then:

```
g/thing/s/x/y/\
s/a/b/
```

is sufficient. The ' \' signals g that the set of commands continues on the next line; it terminates on the first line that does not end with ' \'. You can't use a substitute command to insert a newline within a g command.

Watch out for the command:

```
g/x/s//y/\
s/a/b/
```

which does ***not*** work as you expect. The remembered pattern is the last pattern that was actually executed, so sometimes it will be 'x' (as expected), and sometimes it will be 'a' (not expected). You must spell it out, like this:

```
g/x/s/x/y/\
s/a/b/
```

It is also possible to execute `a`, `c` and `i` commands under a global command; as with other multi-line constructions, all that is needed is to add an `\` at the end of each line except the last. Thus to add a `.nf` and `.sp` command before each `.EQ` line, type:

```
g/^\.EQ/i\
.nf\
.sp
```

You do not need a final line containing a '.' to terminate the `i` command, unless you are using further commands under the global command. On the other hand, it does no harm to put it in either.

4.5. Special Characters

Certain characters have unexpected meanings when they occur in the left side of a substitute command, or in a search for a particular line. You may have noticed that things just don't work right when you use some characters like '.', `*`, `$`, and others in context searches and with the substitute command. These special characters are called ***metacharacters***. Basically, `ed` treats these characters as special, with special meanings. For instance, in a context search or the first string of the substitute command only, '.' means 'any character,' not a period, so:

```
/x.y/
```

means 'a line with an 'x', ***any character***, and a 'y' ', ***not*** just 'a line with an 'x', a period, and a 'y'.' A complete list of the special characters is:

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

4.5.1. Matching Anything — the Dot '.'

Use the 'dot' metacharacter '.' to match any single character. For example, to find any line where 'x' and 'y' occur separated by a single character, type:

```
/x.y/
```

You may get any of:

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

and so on.

Since '.' matches a single character, it gives you a way to deal with funny characters that `l` displays. Suppose you have a line that, when displayed with the `l` command, appears as:

```
....    th07is    ....
```

and you want to get rid of the 07 (which represents the bell character, by the way).

The most obvious solution is to try:

```
s/07//
```

but this will fail. (Try it.) The brute force solution, which most people would now take, is to re-type the entire line. This is guaranteed, and is actually quite a reasonable tactic if the line in question isn't too big, but for a very long line, re-typing is a bore. This is where the metacharacter '.' comes in handy. Since '07' really represents a single character, if we say:

```
s/th.is/this/
```

the job is done. The '.' matches the mysterious character between the 'h' and the 'i', ***whatever it is***.

Bear in mind that since '.' matches any single character, the command:

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

converts the first character on a line into a comma (,), which very often is not what you intended.

As is true of many characters in `ed`, the '.' has several meanings, depending on its context. This line shows all three:

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

The first '.' is a line number, the number of the line we are editing, which is called 'line dot'. The second '.' is a metacharacter that matches any single character on that line. The third '.' is the only one that really is an honest literal period. On the ***right*** side of a substitution, '.' is not special. If you apply this command to the line:

```
Now is the time.
```

the result will be:

```
.s/././
.ow is the time.
```

which is probably not what you intended.

4.5.2. Specifying Any Character — the Backslash '\'

The backslash character '\' is special to `ed` as noted in the description of the ampersand. For safety's sake, avoid the backslash where possible. If you have to use one of the special characters in a substitute command, you can turn off its magic meaning temporarily by preceding it with the backslash. Thus:

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

changes '\.*' into 'backslash dot star'.

Since a period means 'any character', the question naturally arises of what to do when you really want a period. For example, how do you convert the line:

```
Now is the time.
```

into:

```
Now is the time?
```

Use the backslash ' \' here as well to turn off any special meaning that the next character might have; in particular, ' \.' converts the '.' from a 'match anything' into a period, so you can use it to replace the period in 'Now is the time.', type:

```
s/\./?/p
Now is the time?
```

ed treats the pair of characters '\.' as a single real period.

You can also use the backslash when searching for lines that contain a special character. Suppose you are looking for a line that contains:

```
.PP
```

The search for .PP finds:

```
/.PP/
THE APPLICATION OF ...
```

because the '.' matches the letter 'A'. But if you say:

```
/\.PP/
```

you will find only lines that contain .PP.

Consider finding a line that contains a backslash. The search:

```
/\/
```

won't work, because the ' \' isn't a literal ' \', but instead means that the second '/' no longer delimits the search. But by preceding a backslash with another one, you can search for a literal backslash. Thus:

```
/\\/
```

does work. Similarly, you can search for a forward slash '/' with:

```
/\//
```

The backslash turns off the meaning of the immediately following '/' so that it doesn't terminate the /.../ construction prematurely.

As an exercise, before reading further, find two substitute commands that each convert the line:

```
\x\.\y
```

into the line:

```
\x\y
```

Here are several solutions; verify that each works as advertised.

```
s/\\\./
s/x../x/
s/..y/y/
```

Here are a couple of miscellaneous notes about backslashes and special characters. First, you can use any character to delimit the pieces of an s command: there is nothing sacred about

slashes. But you must use slashes for context searching. For instance, in a line that contains a lot of slashes already, like:

```
//exec //sys.fort.go // etc...
```

you could use a colon as the delimiter — to delete all the slashes, type:

```
s:/::g
```

When you are adding text with `a` or `i` or `c`, the backslash is not special, and you should only put in one backslash for each one you really want.

4.5.3. Specifying the End of Line — the Dollar Sign `$`

The dollar-sign, `$`, denotes the end of a line:

```
/string$/
```

only finds an occurrence of 'string' that is at the end of some line. This implies, of course, that:

```
/^string$/
```

finds a line that contains just 'string', and:

```
/^.$/
```

finds a line containing exactly one character.

As an obvious use, suppose you have the line:

```
Now is the
```

and you wish to add the word 'time' to the end. Use the `$` like this:

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

Notice that a space is needed before 'time' in the substitute command, or you will get:

```
Now is thetime
```

As another example, replace the second comma in a line with a period without altering the first comma. Type:

```
s/,$/./p
Now is the time, for all good men,
```

The `$` sign here specifies the comma at the end of the sentence. Without it, of course, `s` operates on the first comma to produce:

```
s/,/./p
Now is the time. for all good men,
```

As another example, to convert:

```
Now is the time.
```

into:

```
Now is the time?
```

as you did earlier, you can use:

```
s/.$/?/p
Now is the time?
```

Like '.', the $ has multiple meanings depending on context. In the line:

```
$s/$/$/
```

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.

4.5.4. Specifying the Beginning of the Line — the Circumflex ^

The circumflex ^ signifies the beginning of a line. Thus:

```
/^string/
string
```

finds 'string' only if it is at the beginning of a line, but not:

```
the string...
```

You can also use ^ to insert something at the beginning of a line. For example, to place a space at the beginning of the current line, type:

```
s/^/ /
```

You can combine metacharacters. To search for a line that contains ***only*** the characters .PP by typing:

```
/^\.PP$/
.
```

*4.5.5. Matching Anything — the Star **

Suppose you have a line that looks like this:

text x y *text*

where *text* stands for lots of text, and there are some indeterminate number of spaces between the 'x' and the 'y'. Suppose the job is to replace all the spaces between 'x' and 'y' by a single space. The line is too long to retype, and there are too many spaces to count. What now?

This is where the metacharacter * comes in handy. A character followed by a star stands for as many consecutive occurrences of that character as possible. To refer to all the spaces at once, say:

```
s/x *y/x y/
```

The construction * means 'as many spaces as possible'. Thus x *y means 'an x, as many spaces as possible, then a y'.

You can use the star with any character, not just the space. If the original example was instead:

text x--------y *text*

then you can replace all − signs by a single space with the command:

```
s/x.*y/x y/
```

Finally, suppose that the line was:

text x.....y *text*

Can you see what trap lies in wait for the unwary? What will happen if you blindly type:

```
s/x.*y/x y/
```

The answer, naturally, is that it depends. If there are no other x's or y's on the line, then everything works, but it's blind luck, not good management. Remember that '.' matches ***any*** single character. Then '.*' matches as many single characters as possible, and unless you're careful, it can eat up a lot more of the line than you expected. If the line was, for example, like this:

text x *text* x.....y *text* y *text*

then saying:

```
s/x.*y/x y/
```

takes everything from the ***first*** 'x' to the ***last*** 'y'. In this example, this is more than you wanted.

The solution, of course, is to turn off the special meaning of '.' with `\.`:

```
s/x\.*y/x y/
```

Now everything works, for `\.*` means 'as many ***periods*** as possible'.

The dot is useful in conjunction with `*`, a repetition character; `a*` is a shorthand for 'any number of 'a' s', so `.*` matches any number of anythings. Use this like:

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

which changes an entire line, or:

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

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

There are times when the pattern `.*` is exactly what you want. For example, use:

```
Now is the time for all good men ....
s/ for.*/./p
Now is the time.
```

The `.*` replaces all of the characters from the space before the word 'for' with a dot. The string 'Now is the time.' is the result in this example.

There are a couple of additional pitfalls associated with `*` that you should be aware of. First note that 'as many as possible' means *zero* or more. The fact that zero is a legitimate possibility is sometimes rather surprising. For example, if your line contained:

text xy *text* x y *text*

and you said:

```
s/x *y/x y/
```

the ***first*** 'xy' matches this pattern, for it consists of an 'x', zero spaces, and a 'y'. The result is that the substitute acts on the first 'xy', and does not touch the later one that actually contains some intervening spaces.

The way around this, if it matters, is to specify a pattern like:

```
/x␣␣*y/
```

where ␣ represents a blank. This describes 'an x, a space, then as many more spaces as possible, then a y', in other words, one or more spaces.

The other startling behavior of * is again related to the fact that zero is a legitimate number of occurrences of something followed by a star. The following command does not produce what was intended:

```
abcdef
s/x*/y/g
p
yaybycydyeyfy
```

The reason for this behavior again, is that zero is a legal number of matches, and there are no x's at the beginning of the line (so that gets converted into a 'y'), nor between the 'a' and the 'b' (so that gets converted into a 'y'), nor ... and so on. Make sure you really want zero matches; if not, in this case write:

```
s/xx*/y/g
```

xx* is 'one or more 'x's'.

4.5.6. Character Classes — Brackets [ ]

The [and] brackets form 'character classes'. Any characters can appear within a character class, and just to confuse the issue, there are essentially no special characters inside the brackets; even the backslash doesn't have a special meaning. For example, to match any single digit, use:

```
/[0123456789]/
```

Any one of the characters inside the braces will cause a match. It is a nuisance to have to spell out the digits, so you can abbreviate them as [0—9]. Similarly, [a—z] stands for the lower-case letters, and [A—Z] for upper case.

Suppose that you want to delete any numbers that appear at the beginning of all lines of a file. You might first think of trying a series of commands like:

```
1,$s/^1*//
1,$s/^2*//
1,$s/^3*//
```

and so on, but this is clearly going to take forever if the numbers are at all long. Unless you want to repeat the commands over and over until all numbers are gone, you must get all the digits on one pass. This is the purpose of the brackets [and].

Another example: To match zero or more digits (an entire number), and to delete all digits from the beginning of all lines, type:

```
1,$s/^[0123456789]*//
```

To search for special characters, for example, you can say:

```
/[.\$^[ ]/
```

Within [...], the [is not special. To get a] into a character class, make it the first character.

As a final frill on character classes, you can specify a class that means 'none of the following characters'. To do this, begin the class with a caret (^) to stand for 'any character ***except*** a digit':

```
[^0-9]
```

Thus you might find the first line that does not begin with a tab or space by a search like:

```
/^[^(space)(tab)]/
```

Within a character class, the circumflex has a special meaning only if it occurs at the beginning. Just to convince yourself, verify that to find a line that doesn't begin with a circumflex, you type:

```
/^[^^]/
```

4.6. Cutting and Pasting with the Editor

ed has commands for manipulating individual lines or groups of lines in files.

4.6.1. Moving Lines Around

There are several ways to move text around in a file.

4.6.2. Moving Text Around — the Move Command m

Use the ***move*** command **m** for cutting and pasting — you can move a group of lines from one place to another in the buffer. Suppose you want to put the first three lines of the buffer at the end instead. You could do it by saying:

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

This is the brute force way; that is, you write the paragraph into a temporary file, read in the temporary file at the end, and then delete it from its current position. As another example, consider:

```
.,/^\.PP/-w temp
.,//-d
$r temp
```

That is, from where you are now ('.') until one line before the next .PP (/^\.PP/–), write into ***temp***. Then delete the same lines. Finally, read in ***temp*** at the end.

But you can do it a lot easier with **m**, so you can do a whole operation at one crack.

```
1,3m$
```

The general case is:

start line, end line* m *after this line

Notice that there is a third line to be specified — the place where the moved stuff gets put.

If you try:

```
1,5m3
?
```

ed reminds you that you can't do this.

The m command is like many other ed commands in that it takes up to two line numbers in front that tell what lines are to be moved. It is also ***followed*** by a line number that tells where the lines are to go. Thus:

line1, line2 **m** *line3*

says to move all the lines between 'line1' and 'line2' after 'line3'. Naturally, any of 'line1' etc., can be patterns between slashes, dollar signs, or other ways to specify lines.

Of course you can specify the lines to be moved by context searches; if you had:

First paragraph
. . .
end of first paragraph.
Second paragraph
. . .
end of second paragraph.

you could reverse the two paragraphs like this:

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

Notice the -1: the moved text goes ***after*** the line mentioned. Dot gets set to the last line moved. Suppose you want to move a paragraph from its present position in a paper to the end. How would you do it? As a hint, suppose each paragraph in the paper begins with the formatting command .PP. Think about it and write down the details before reading on.

Suppose again that you're sitting at the first line of the paragraph. Then you can say:

```
.,/^\.PP/-m$
```

That's all.

As another example of a frequent operation, you can reverse the order of two adjacent lines by moving the first one after the second. Suppose that you are positioned at the first. Then, to move line dot to one line after line dot, type:

```
m+
```

If you are positioned on the second line, and want to do the reverse, type:

```
m--
```

As you can see, m is more succinct and direct than writing, deleting and re-reading. When is brute force better? This is a matter of personal taste — do what you have most confidence in. The main difficulty with m is that if you use patterns to specify both the lines you are moving and the target, you have to take care that you specify them properly, or you may well not move the lines you thought you did. The result of a botched m command can be a mess. Doing the job a step at a time makes it easier for you to verify at each step that you accomplished what you wanted to. It's also a good idea to use a w command before doing anything complicated; then if you goof, it's easy to back up to where you were.

4.6.3. Substituting Newlines

You can split a single line into two or more shorter lines by 'substituting in a newline'. As the simplest example, suppose a line has gotten unmanageably long because of editing or merely because it was unwisely typed. If it looks like:

```
text    xy    text
```

you can break it between the 'x' and the 'y' like this:

```
s/xy/x\
y/
```

This is actually a single command, although it is typed on two lines. Bearing in mind that '\' turns off special meanings, it seems relatively intuitive that a '\' at the end of a line would make the newline there no longer special.

You can in fact make a single line into several lines with this same mechanism. As a large example, consider underlining the word 'very' in a long line by splitting 'very' onto a separate line, and preceding it by the `nroff` formatting command `.ul`.

```
text  a very big  text
```

To convert the line into four shorter lines, preceding the word 'very' by the line `.ul`, and eliminating the spaces around the 'very', all at the same time, type:

```
s/ very /\
.ul\
very\
/
```

When a newline is substituted in, dot is left pointing at the last line created.

4.6.4. Joining Lines — the Join Command `j`

You may also join lines together, but use the ***join*** command `j` for this instead of `s`. Given the lines:

```
Now is
 the time
```

and supposing that dot is set to the first of them, then the command:

```
j
```

joins them together. No blanks are added, which is why we carefully showed a blank at the beginning of the second line.

All by itself, a `j` command joins line dot to line dot+1, but any contiguous set of lines can be joined. Just specify the starting and ending line numbers. For example:

```
1,$jp
```

joins all the lines into one big one and displays it.

4.6.5. Rearranging a Line with \(... \)

Skip this section if this is the first time you're reading this chapter. Recall that & stands for whatever was matched by the left side of an s command. In much the same way you can capture separate pieces of what was matched; the only difference is that you have to specify on the left side just what pieces you're interested in.

Suppose, for instance, that you have a file of lines that consist of names in the form:

```
Smith, A. B.
Jones, C.
```

and so on, and you want the initials to precede the name, as in:

```
A. B. Smith
C. Jones
```

It is possible to do this with a series of editing commands, but it is tedious and error-prone. (It is instructive to figure out how it is done, though.)

The alternative is to 'tag' the pieces of the pattern, in this case, the last name, and the initials, and then rearrange the pieces. On the left side of a substitution, if part of the pattern is enclosed between \(and \), whatever matched that part is remembered, and available for use on the right side. On the right side, the symbol \1 refers to whatever matched the first \(...\) pair, \2 to the second \(...\), and so on.

The command:

```
1,$s/^\([^,]*\), *\(.*\)/\2 \1/
```

although hard to read, does the job. The first \(...\) matches the last name, which is any string up to the comma; this is referred to on the right side with \1. The second \(...\) is whatever follows the comma and any spaces, and is referred to as '\2'.

Of course, with any editing sequence this complicated, it's foolhardy to simply run it and hope. The global commands g and v provide a way for you to display exactly those lines which were affected by the substitute command, and thus verify that it did what you wanted in all cases.

4.6.6. Marking a Line — the Mark Command k

You can mark a line with a particular name so you can refer to it later by name, regardless of its actual line number. This can be handy for moving lines, and for keeping track of them as they move. The *mark* command is k. To mark the current line with the name x, use:

```
kx
```

If a line number precedes the k, that line is marked. The mark name must be a single lowercase 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.

4.6.7. Copying Lines — the Transfer Command t

We mentioned earlier the idea of saving a line that was hard to type or used often, to cut down on typing time. Of course this can be more than one line, in which case the saving is presumably even greater.

ed provides another command, called t (*transfer*) for making a copy of a group of one or more lines at any point. This is often easier than writing and reading.

The t command is identical to m, except that instead of moving lines, it simply duplicates them at the place you named. Thus, to duplicate the entire contents that you are editing, use:

```
1,$t$
```

A more common use for t is for creating a series of lines that differ only slightly. For example, you can say:

```
a
.......  x  ........ (long line)
.
t.          (make a copy)
s/x/y/      (change it a bit)
t.          (make third copy)
s/y/z/ (change it a bit)
```

and so on.

4.7. Escaping to the Shell with !

Sometimes it is convenient to be able to temporarily escape from the editor to use some Shell command without leaving the editor. Use the ! (escape) command to do this.

To suspend your current editing state and execute the shell command you asked for, type:

```
!any shell command
!
```

When the command finishes, ed will signal you by displaying another !; at that point, you can resume editing.

You can really do *any* shell command, including another ed. This is quite common, in fact. In this case, you can even do another !.

4.8. Supporting Tools

There are several tools and techniques that go along with the editor, all of which are relatively easy once you know how ed works, because they are all based on the editor. This section gives some fairly cursory examples of these tools, more to indicate their existence than to provide a complete tutorial. For more information on each, refer to the ***Commands Reference Manual for the Sun Workstation***.

4.8.1. Editing Scripts

If you have a fairly complicated set of editing operations to do on a whole set of files, the easiest thing to do is to make up a 'script', that is, a file that contains the operations you want to perform, and then apply this script to each file in turn.

For example, suppose you want to change every 'SUN' to 'Sun' and every 'SYSTEM' to 'System' in a large number of files. Then put into a file, which we'll call ***changes***, the lines:

```
g/SUN/s//Sun/g
g/SYSTEM/s//System/g
w
q
```

Now you can say:

```
hostname% ed file1 <script
hostname% ed file2 <script
...
```

This causes ed to take its commands from the prepared script called ***changes***. Notice that you have to plan the whole job in advance.

And of course by using the Sun UNIX† command interpreter, the shell, you can cycle through a set of files automatically, with varying degrees of ease.

4.8.2. Matching Patterns with grep

Sometimes you want to find all occurrences of some word or pattern in a set of files, to edit them or perhaps just to verify their presence or absence. You can edit each file separately and look for the pattern of interest, but if there are many files, this can get very tedious, and if the files are really big, it may be impossible because of limits in ed.

The program grep gets around these limitations. The search patterns that are described in this chapter are often called 'regular expressions', and 'grep' stands for 'general regular expression, print.' That describes exactly what grep does — it displays every line in a set of files that contains a particular pattern. Thus, to find 'thing' wherever it occurs in any of the files ***file1***, ***file2***, etc., type:

```
hostname% grep 'thing' file1 file2 file3 ...
hostname%
```

grep also indicates the file in which the line was found, so you can later edit it if you like.

The pattern represented by 'thing' can be any pattern you can use in the editor, since grep and ed use exactly the same mechanism for pattern searching. It is wisest always to enclose the pattern in the single quotes '...' if it contains any non-alphabetic characters, since many such characters also mean something special to the Sun UNIX command interpreter, the shell. If you don't quote them, the command interpreter will try to interpret them before grep gets a chance.

There is also a way to find lines that ***do not*** contain a pattern:

---

[19] † UNIX is a trademark of Bell Laboratories.

```
hostname% grep -v 'thing' file1 file2 ...
hostname%
```

finds all lines that don't contain 'thing'. The **-v** must occur in the position shown. Given `grep` and `grep -v`, it is possible to do things like selecting all lines that contain some combination of patterns. For example, to get all lines that contain 'x' but not 'y', use:

```
hostname% grep x file... | grep -v y
hostname%
```

The notation | is a 'pipe', which causes the output of the first command to be used as input to the second command; see the ***Beginner's Guide to the Sun Workstation*** for an introduction to 'piping.' See the ***Commands Reference Manual for the Sun Workstation*** for details on `grep`.

4.9. Summary of Commands and Line Numbers

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

`a` Append, that is, add lines to the buffer at line dot, unless a different line is specified. Type a '.' on a new line to terminate appending. Dot is set to the last line appended.

`c` Change the specified lines to the new text that follows. Type a '.' as with `a` to terminate the change. If no lines are specified, replace line dot. Dot is set to last line changed.

`d` Delete the lines specified. If none is specified, delete line dot. Dot is set to the first undeleted line, unless '$' is deleted, in which case dot is set to '$'.

`e` Edit new file. Any previous contents of the buffer are thrown away, so use a `w` beforehand.

`f` Print remembered filename. If a name follows `f` the remembered name will be set to it.

`g` The command:

g/---/commands

executes the commands on those lines that contain ' --- ', which can be any context search expression.

`i` Insert lines before specified line (or dot) until a '.' is typed on a new line. Dot is set to last line inserted.

`m` Move lines specified to after the line named after `m`. Dot is set to the last line moved.

`p` Display specified lines. If none is specified, display line dot. A single line number is equivalent to ***line-number*** `p`. Type a single RETURN to show `.+1`, the next line.

`q` Quit `ed`. This wipes out all text in buffer if you give it twice in a row without first giving a `w` command.

`r` Read a file into the buffer at the end unless an address is specified. Dot is set to the last line read.

s The command:

s/string1/string2/

substitutes the characters 'string2' into 'string1' in the specified lines. If no lines are specified, make the substitution in line dot. Dot is set to last line in which a substitution took place, which means that if no substitution took place, dot is not changed. An **s** changes only the first occurrence of 'string1' on a line; to change all of them, type a **g** after the final slash.

v The command:

v/---/*commands*

executes ***commands*** on those lines that ***do not*** contain ' --- '.

w Write out buffer into a file. Dot is not changed.

.= Show value of dot (current line number). An = by itself shows the value of '$.' (number of the last line in the buffer).

! The line:

!*command*

executes ***command*** as a Sun UNIX shell command.

/-----/ Context search. Search for next line that contains this string of characters and display it. Dot is set to the line where string was found. Search starts at '.+1', wraps around from '$' to 1, and continues to dot, if necessary.

?-----? Context search in reverse direction. Start search at '.−1', scan to 1, wrap around to '$.'.

Chapter 5

Using sed, the Stream Text Editor

This chapter[20] describes **sed**, the non-interactive context or ***stream*** editor. Use **sed** for editing files too large for comfortable interactive editing, editing any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode, and performing multiple global editing functions efficiently in one pass through the input. Because the default mode is to apply edit commands globally, and because its output is to the standard output, your workstation or terminal screen, **sed** is good for making changes of a transient nature, rather than permanent modifications to a file.

You can create a complicated editing script separately and use it as a command file. For complex edits, this saves considerable typing, and its attendant errors. Running **sed** from a command file is much more efficient than any interactive editor even if that editor can be driven by a prewritten script.

Whereas the **ed** editor copies your original file into a buffer, **sed** does not use temporary files so you can edit any size file. The only space requirement is that the input and output fit simultaneously into the available second storage. Additionally, **ed** lets you explore the text in whatever order you want, while **sed** works on your file from beginning to end, and allows you no choice of edit commands once you have started it. Basically **sed** passes some data through a set of transformations called editor ***functions***.

By default **sed** copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. You can modify this behavior by adding a command-line option; see the "Command Options" section below.

As a lineal descendant of the **ed** editor, **sed** recognizes basically the same regular expressions as **ed**. The range of pattern matches is called the ***pattern space***. Ordinarily, the pattern space is one line of text, but you can read more than one line into the pattern space if necessary. But because of the differences between interactive and non-interactive operation, **ed** and **sed** are different enough that even experienced **ed** users should read this chapter. You cannot use relative addressing with **sed** as you can with an interactive editor because **sed** operates a line at a time. **sed** also does not give you any immediate verification that a command has done what was intended.

Refer to the chapter on "Using the **ed** Line Editor" for more information on **ed** and to the descriptions of **sed** and **ed** in the *Commands Reference Manual for the Sun Workstation*.

---

[20] The material in this chapter is derived from *Sed — a Non-Interactive Text Editor*, L.E. McMahon, Bell Laboratories, Murray Hill, New Jersey.

5.1. Using sed

The general format of an editing command is:

```
hostname% sed [line1[,line2]] function [arguments]
```

There is an optional line address, or two line addresses separated by a comma, a single-letter edit function, followed by other arguments, which may be required or optional, depending on which function you use. See the section "Specifying Lines for Editing" for the format of line addresses. Any number of blanks or tabs may separate the line addresses from the function. **sed** ignores tab characters and spaces at the beginning of lines. The function must be present; the available commands are discussed in the "Functions" section under each individual function name. You can either put the edit commands on the **sed** command line or put the commands in a file, which is then applied to the file you want to edit. If the commands are few and simple, put them on the **sed** command line. For example, assume the following input text in a file called ***kubla***:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

Let's copy the first two lines of input as a simple example:

```
hostname% sed 2q kubla
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

As another example, suppose that you want to change the 'Khan' to 'KHAN.' Then the command:

```
hostname% sed  s/Khan/KHAN/g kubla
```

applies the command 's/Khan/KAN/' to all lines from ***kubla*** and copies all lines to the standard output. The advantage of using **sed** in such a case is that you can use it 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:

```
hostname% sed  -f  cmdfile input-files...
```

5.1.1. Command Options

sed has three options that modify **sed**'s action. If you invoke **sed** with the **-f** (file) option, the edit commands are taken from a file. For example:

```
hostname% sed -f edcomds oldfile > newfile
hostname%
```

The name of the file containing the edit commands must immediately follow the **-f** option. Here, the edit commands in the ***edcomds*** file are applied to the file ***oldfile***, and the standard output is redirected to ***newfile***.

You use the **-e** (edit) option to place editing commands directly on the **sed** command line. If you are only using one edit command, you can omit the **-e**, but we include it in the example below for instructive purposes. For example, to delete a line containing the string 'Khan' from *kubla*, you type:

```
hostname% sed -e /Khan/d kubla > newkubla
hostname%
```

If you put more than one edit command on the **sed** command line, each one must be preceded by **-e**. For example:

```
hostname% sed -e /Khan/d -e s/decree/DECREE/ newkubla
hostname%
```

You can also use both the **-e** and the **-f** options at the same time.

sed normally copies all input lines that are changed by the edit operation to the output. If you want to suppress this normal output, and have only specific lines appear on the output, use the **-n** option with the **p** (print) flag. For example:

```
hostname% sed -n -e s/to/by/p kubla
Through caverns measureless by man
Down by a sunless sea.
hostname%
```

As a quick reference, these options are:

-f Use the next argument as a filename; the file should contain one editing command to a line.

-e Use the next argument as an editing command.

-n Send only those lines to the output specified by **p** functions or **p** functions after **substitute** functions (see the "Input-Output Functions" section).

5.2. Editing Commands Application Order

Before any editing is done (in fact, before any input file is even opened), all the editing commands are compiled into a moderately efficient form for execution when the commands are actually applied to lines of the input file. The commands are compiled in the order in which they are encountered; this is generally the order in which they will be attempted at execution time. The commands are applied one at a time; the input to each command is the output of all preceding commands.

You can change the default linear order of application of editing commands by the flow-of-control commands, **t** and **b** (see the "Flow-of-Control Functions" section). Even when you change the order of application by these commands, it is still true that the input line to any command is the output of any previously applied command.

5.3. Specifying Lines for Editing

Use addresses to select lines in the input file(s) to apply the editing commands to. Addresses may be either line numbers or context addresses.

Group one address or address-pair with curly braces '{ }' to control the application of a group of commands. See the "Flow-of-Control Functions" section for more on this.

5.3.1. Line-number Addresses

A line number is a decimal integer. As each line is read from the input, a line-number counter is incremented; a line-number address matches or 'selects' the input line which causes the internal counter to equal the address line-number. The counter runs cumulatively through multiple input files; it is not reset when a new input file is opened.

As a special case, the character `$` matches the last line of the last input file.

5.3.2. Context Addresses

A context address is a pattern or ***regular expression*** enclosed in slashes (/). **sed** recognizes the regular expressions that are constructed as follows:

ordinary character
: An ordinary character (not one of those discussed below) is a regular expression, and matches that character.

`^`
: A circumflex `^` at the beginning of a regular expression matches the null character at the beginning of a line.

`$`
: A dollar-sign `$` at the end of a regular expression matches the null character at the end of a line.

`\n`
: The characters backslash and en `\n` match an embedded newline character, but not the newline at the end of the pattern space.

`.`
: A period `.` matches any character except the terminal newline of the pattern space.

`*`
: A regular expression followed by an asterisk '*' matches any number (including 0) of adjacent occurrences of the regular expression it follows.

[character string]
: A string of characters in square brackets `[ ]` matches any character in the string, and no others. If, however, the first character of the string is a circumflex `^`, the regular expression matches any character ***except*** the characters in the string and the terminal newline of the pattern space.

concatenation
: A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.

`\(  \)`
: A regular expression between the sequences `\(` and `\)` is identical in effect to the unadorned regular expression, but has side-effects which are described in the section entitled "The Substitute Function **s**" and immediately below.

`\`*d*
: This stands for the same string of characters matched by an expression enclosed in `\(` and `\)` earlier in the same pattern. Here *d* is a single digit; the string specified is that beginning with the *d*th occurrence of `\(` counting from the left. For example, the expression `^\(.*\)\1` matches a line beginning with two repeated occurrences of the same string.

null The null regular expression standing alone (such as, //) is equivalent to the last regular expression compiled.

To use one of the special characters (^ $. * [ ] \ /) as a literal, that is, to match an occurrence of itself in the input, precede the special character by a backslash \.

For a context address to 'match' the input requires that the whole pattern within the address match some portion of the pattern space.

5.3.3. Number of Addresses

The commands described in the "Functions" section can have 0, 1, or 2 addresses. Specifying more than the maximum number of addresses allowed is an error. If a command has no addresses, it is applied to every line in the input. If a command has one address, it is applied to all lines that match that address. If a command has two addresses, it is applied to the inclusive range defined by those two addresses.

The command is applied to the first line that matches the first address, and to all subsequent lines until and including the first subsequent line which matches the second address. Then an attempt is made on subsequent lines to again match the first address, and the process is repeated. A comma separates two addresses.

For example:

```
/an/                        matches lines 1, 3, 4 in our sample kubla file
In Xanadu did Kubla Khan
Where Alph, the sacred river, ran
Through caverns measureless to man

/an.*an/                    matches line 1
In Xanadu did Kubla Khan

/^an/                       matches no lines


/./                         matches all lines
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.

/\./                        matches line 5
Down to a sunless sea.

/r*an/                      matches lines 1,3, 4 (number = zero!)
In Xanadu did Kubla Khan
Where Alph, the sacred river, ran
Through caverns measureless to man

/\(an\).*\1/                matches line 1
In Xanadu did Kubla Khan
```

5.4. Functions

All functions are named by a single character. In the following summary, the maximum number of allowable addresses is enclosed in parentheses, followed by the single character function name and possible arguments in italics. The summary provides an expanded English translation of the single-character name, and a description of what each function does.

5.4.1. Whole Line Oriented Functions

The functions that operate on a whole line of input text are as follows:

(2) **d** Delete lines. The **d** function deletes from the file all those lines matched by its address(es); that is, it does not write the indicated lines to the output, No further commands are attempted on a deleted line; as soon as the **d** function is executed, a new line is read from the input, and the list of editing commands is re-started from the beginning on the new line.

(2) **n** Next line. The **n** function reads the next line from the input, replacing the current line. The current line is written to the output if it should be. The list of editing commands is continued following the **n** command.

(1) **a**

text Append lines. The **a** function writes the argument ***text*** to the output after the line matched by its address. The **a** function is inherently multi-line; **a** must appear at the end of a line, and ***text*** may contain any number of lines. To preserve the one command to a line, the interior newlines must be hidden by a backslash character (\) immediately preceding the newline. The ***text*** argument is terminated by the first unhidden newline (the first one not immediately preceded by backslash). Once an **a** function is successfully executed, ***text*** will be written to the output regardless of what later commands do to the line that triggered it. The triggering line may be deleted entirely; ***text*** will still be written to the output. The ***text*** is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter.

(1) **i**

text Insert lines. The **i** function behaves identically to the **a** function, except that ***text*** is written to the output ***before*** the matched line. All other comments about the **a** function apply to the **i** function as well.

(2) **c**

text Change lines. The **c** function deletes the lines selected by its address(es), and replaces them with the lines in ***text***. Like **a** and **i**, put a newline hidden by a backslash after **c**; interior new lines in ***text*** must also be hidden by backslashes. The **c** function may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of ***text*** is written to the output, ***not*** one copy per line deleted. As with **a** and **i**, ***text*** is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter.

No further commands are attempted on a line deleted by a **c** function.

If text is appended after a line by `a` or `r` functions, and the line is subsequently changed, the text inserted by the `c` function will be placed ***before*** the text of the `a` or `r` functions. See the section "Multiple Input-line Functions" later in this chapter for a description of the `r` function.

Note: Leading blanks and tabs are not displayed in the output produced by these functions. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash does not appear in the output.

For example, put the following list of editing commands in a file called *Xkubla*:

```
hostname% cat > Xkubla
n
a\
XXXX
d
^D
hostname% sed -f Xkubla kubla
In Xanadu did Kubla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
hostname%
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n
i\
XXXX
d
```

or

```
n
c\
XXXX
```

5.4.2. The Substitute Function `s`

The `s` (substitute) function changes parts of lines selected by a context search within the line. The standard format is the same as the `ed` substitute command:

(2) **`s`** ***pattern replacement flags***

The `s` function replaces ***part*** of a line, selected by ***pattern***, with ***replacement***. It can best be read 'Substitute for ***pattern, replacement***.'

The ***pattern*** argument contains a pattern, exactly like the patterns described in the "Specifying Lines for Editing" section. The only difference between ***pattern*** and a context address is that the context address must be delimited by slash (/) characters; you can delimit ***pattern*** by any character other than space or newline.

By default, only the first string matched by ***pattern*** is replaced. See the `g` flag below.

The *replacement* argument begins immediately after the second delimiting character of *pattern*, and must be followed immediately by another instance of the delimiting character. Thus there are exactly *three* instances of the delimiting character.

The *replacement* is not a pattern, and the characters which are special in patterns do not have special meaning in *replacement*. Instead, other characters are special:

`&` Is replaced by the string matched by *pattern*.

`\`*d* Is replaced by the *d*th substring matched by parts of *pattern* enclosed in `\(` and `\)` where *d* is a single digit. If nested substrings occur in *pattern*, the *d*th is determined by counting opening delimiters ('\(').

As in patterns, you can make the special characters (`&`, `+`, and `\`) literal by preceding them with a backslash (`\`).

The *flags* argument may contain the following flags:

`g` Substitute *replacement* for all (non-overlapping) instances of *pattern* in the line. After a successful substitution, the scan for the next instance of *pattern* begins just after the end of the inserted characters; characters put into the line from *replacement* are not rescanned.

`p` Print or 'display' the line if a successful replacement was done. The `p` flag writes the line to the output if and only if a substitution was actually made by the `s` function. Notice that if several `s` functions, each followed by a `p` flag, successfully substitute in the same input line, multiple copies of the line will be written to the output: one for each successful substitution.

`w` *filename*
Write the line to a file if a successful replacement was done. The `w` flag writes lines which are actually substituted by the `s` function to a file named by *filename*. If *filename* exists before `sed` is run, it is overwritten; if not, it is created. A single space must separate `w` and *filename*. The possibilities of multiple, somewhat different copies of one input line being written are the same as for `p`. You can specify a maximum of 10 different filenames after `w` flags and `w` functions (see below), combined.

For example, applying the following command to the the *kubla* file produces on the standard output:

```
hostname% sed -e "s/to/by/w changes" kubla
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

Note that if the edit command contains spaces, you must enclose it with quotes.

It also creates a new file called *changes* that contains only the lines changed as you can see using the `more` command:

```
hostname% more changes
Through caverns measureless by man
Down by a sunless sea.
hostname%
```

If the nocopy option -n is in effect, you see those lines that are changed:

```
hostname% sed -e "s/[.,;?:]/*P&*/gp" -n kubla
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
hostname%
```

Finally, to illustrate the effect of the g flag assuming nocopy mode, consider: LS hostname% **sed -e "/X/s/an/AN/p" -n kubla** In XANadu did Kubla Khan hostname%

and the command:

```
hostname% sed -e "/X/s/an/AN/gp" -n kubla
In XANadu did Kubla KhAN
hostname%
```

5.4.3. Input-output Functions

The following functions affect the input and output of text. The maximum number of allowable addresses is in parentheses.

(2) p Print. The print function writes the addressed lines to the standard output file. They are written at the time the p function is encountered, regardless of what succeeding editing commands may do to the lines.

(2) w *filename*
Write to *filename*. The write function writes the addressed lines to the file named by *filename*. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands may do to them. Put only one space between w and *filename*. You can use a maximum of ten different files in write functions and with w flags after s functions, combined.

(1) r *filename*
Read the contents of a file. The read function reads the contents of *filename*, and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address. If you execute r and a functions on the same line, the text from the a functions and the r functions is written to the output in the order that the functions are executed. Put only one space between the r and *filename*. If a file mentioned by a r function cannot be opened, it is considered a null file, not an error, and no diagnostic is displayed.

Note: Since there is a limit to the number of files that can be opened simultaneously, put no more than ten files in w functions or flags; reduce that number by one if any r functions are present. Only one read file is open at one time.

Assume that the file *note1* has the following contents:

```
Note:  Kubla Khan (more properly Kublai Khan; 1216-1294)
was the grandson and most eminent successor of Genghiz
(Chingiz) Khan, and founder of the Mongol dynasty in China.
```

Then the following command reads in *note1* after the line containing 'Kubla':

```
hostname% sed -e "/Kubla/r note1" kubla
In Xanadu did Kubla Khan
Note:  Kubla Khan (more properly Kublai Khan; 1216-1294)
was the grandson and most eminent successor of Genghiz
(Chingiz) Khan, and founder of the Mongol dynasty in China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

5.4.4. Multiple Input-line Functions

Three functions, all spelled with capital letters, deal specially with ***pattern spaces*** containing embedded newlines; they are intended principally to provide pattern matches across lines in the input. A pattern space is the range of pattern matches. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the `N` function described below.

The maximum number of allowable addresses is enclosed in parentheses.

(2) `N` Next line. The next input line is appended to the current line in the pattern space; an embedded newline separates the two input lines. Pattern matches may extend across the embedded newline(s).

(2) `D` Delete first part of the pattern space. Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), read another line from the input. In any case, begin the list of editing commands again from its beginning.

(2) `P` Print or 'display' first part of the pattern space. Print up to and including the first newline in the pattern space.

The `P` and `D` functions are equivalent to their lower-case counterparts if there are no embedded newlines in the pattern space.

5.4.5. Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

(2) `h` Hold pattern space. The `h` function copies the contents of the pattern space into a hold area, destroying the previous contents of the hold area.

(2) `H` Hold pattern space. The `H` function appends the contents of the pattern space to the contents of the hold area; the former and new contents are separated by a newline.

(2) `g` Get contents of hold area. The `g` function copies the contents of the hold area into the pattern space, destroying the previous contents of the pattern space.

(2) `G` Get contents of hold area. The `G` function appends the contents of the hold area to the contents of the pattern space; the former and new contents are separated by a newline.

(2) `x` Exchange. The exchange command interchanges the contents of the pattern space and the hold area.

For example, if you want to add `:In Xanadu` to our standard example, create a file called *test* containing the following commands:

```
1h
1s/ did.*//
1x
G
s/\n/  :/
```

Then run that file on the *kubla* file:

```
hostname% sed -f test kubla
In Xanadu did Kubla Khan  :In Xanadu
A stately pleasure dome decree:  :In Xanadu
Where Alph, the sacred river, ran  :In Xanadu
Through caverns measureless to man  :In Xanadu
Down to a sunless sea.  :In Xanadu
hostname%
```

5.4.6. Flow-of-Control Functions

These functions do not edit the input lines, but control the application of functions to the lines that are addressed.

(2) `!` Called 'Don't', the '`!`' function applies the next command, written on the same line, to all and only those input lines ***not*** selected by the address part.

(2) `{` Grouping. The grouping command '`{`' applies (or does not apply) the next set of commands as a block to the input lines that the addresses of the grouping command select. The first of the commands under control of the grouping command may appear on the same line as the `{` or on the next line.

A matching `}` standing on a line by itself terminates the group of commands. Groups can be nested.

(0) `:` ***label***

Place a label. The label function marks a place in the list of editing commands which may be referred to by `b` and `t` functions. The ***label*** may be any sequence of eight or fewer characters; if two different colon functions have identical labels, a compile time diagnostic will be generated, and no execution attempted.

(2) `b` ***label***

Branch to label. The branch function restarts the sequence of editing commands being applied to the current input line immediately after the place where a colon function with the same ***label*** was encountered. If no colon function with the same label can be found after all the editing commands have been compiled, a compile time diagnostic is produced, and no execution is attempted.

A `b` function with no ***label*** is taken to be a branch to the end of the list of editing commands. Whatever should be done with the current input line is done, and another input line is read. The list of editing commands is restarted from the beginning on the new line.

(2) `t` *label*

Test substitutions. The `t` function tests whether ***any*** successful substitutions have been made on the current input line; if so, it branches to ***label***; if not, it does nothing. Either reading a new input line or executing a `t` function resets the flag which indicates that a successful substitution has occurred.

5.4.7. Miscellaneous Functions

Two additional functions are:

(1) `=` Equals. The `=` function writes to the standard output the line number of the line matched by its address.

(1) `q` Quit. The `q` function writes the current line to the output if it should be, writes any appended or read text, and terminates execution.

Chapter 6

Pattern Scanning and Processing with awk

awk is a utility program that you can program in varying degrees of complexity. awk's basic operation is to search a set of files for patterns based on ***selection criteria***, and to perform specified actions on lines or groups of lines which contain those patterns. Selection criteria can be text patterns or ***regular expressions***. awk makes data selection, transformation operations, information retrieval and text manipulation easy to state and to perform.

Basic awk operation is to scan a set of input lines in order, searching for lines which match any of a set of patterns that you have specified. You can specify an action to be performed on each line that matches the pattern.

awk patterns may include arbitrary Boolean combinations of regular expressions and of relational and arithmetic operators on strings, numbers, fields, variables, and array elements. Actions may include the same pattern-matching constructions as in patterns, as well as arithmetic and string expressions and assignments, `if-else`, `while`, `for` statements, and multiple output streams.

If you are familiar with the grep utility (see the *Commands Reference Manual for the Sun Workstation*), you will recognize the approach, although in awk, the patterns may be more general than in grep, and the actions allowed are more involved than merely displaying the matching line.

As some simple examples to give you the idea, consider a short file called ***sample***, which contains some identifying numbers and system names:

```
125.1303        krypton loghost
125.0x0733      window
125.1313        core
125.19          haley
```

If you want to display the second and first columns of information in that order, use the awk program:

```
hostname% awk '{print $2, $1}' sample
krypton 125.1303
window 125.0x0733
core 125.1313
haley 125.19
```

This is good for reversing columns of tabular material for example. The next program shows all input lines with an a, b, or c in the second field.

---

[20] The material in this chapter is derived from *Awk — A Pattern Scanning and Processing Language*, A. Aho, B.W. Kernighan, P. Weinberger, Bell Laboratories, Murray Hill, New Jersey.

```
hostname% awk '$2 - /a|b|c/' sample
125.1313        core
125.19          haley
```

6.1. Using awk

The general format for using **awk** follows. You execute the **awk** commands in a string that we'll call ***program*** on the set of named ***files***:

```
hostname% awk  program files
```

For example, to display all input lines whose length exceeds 13 characters, use the program:

```
hostname% awk 'length > 13' sample
125.1303        krypton loghost
125.0x0733      window
hostname%
```

In the above example, the ***program*** compares the length of the ***sample*** file lines to the number 13 and displays lines longer than 13 characters.

awk usually takes its program as the first argument. To take a program from a file instead, use the **-f** (file) option. For example, you can put the same statement in a file called ***howlong***, and execute it on ***sample*** with:

```
hostname% awk  -f howlong  hosts
125.1303        krypton loghost
125.0x0733      window
```

You can also execute **awk** on the standard input if there are no files. Put single quotes around the **awk** program because the shell duplicates most of **awk**'s special characters.

6.1.1. Program Structure

A program can consist of just an action to be performed on all lines in a file, as in the ***howlong*** example above. It can also contain a pattern that specifies the lines for the action to operate on. This pattern/action order is represented in **awk** notation by:

```
pattern   {action }
```

In other words, each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action is executed. When all the patterns have been tested, the next line is fetched and the matching starts over.

Either the pattern or the action may be left out, but not both. If there is no action for a pattern, the matching line is simply copied to the output. Thus a line which matches several patterns can be printed several times. If there is no pattern for an action, the action is performed on every input line. A line which doesn't match any pattern is ignored. Since patterns and actions are both optional, you must enclose actions in braces ({action}) to distinguish them from patterns. See more about patterns in the "Specifying Patterns" section later in this chapter.

6.1.2. Records and Fields

awk input is divided into ***records*** terminated by a ***record separator***. The default record separator is a newline, so by default awk processes its input a line at a time. The number of the current record is available in a variable named NR.

Each input record is considered to be divided into ***fields***. Fields are separated by ***field separators***, normally blanks or tabs, but you can change the input field separator, as described in the "Field Variables" section later in this chapter. Fields are referred to as $X where $1 is the first field, $2 the second, and so on as shown above. $0 is the whole input record itself. Fields may be assigned to. The number of fields in the current record is available in a variable named NF.

The variables FS and RS refer to the input field and record separators; you can change them at any time to any single character. You may also use the optional command-line argument -Fc to set FS to any character *c*.

If the record separator is empty, an empty input line is taken as the record separator, and blanks, tabs and newlines are treated as field separators.

The variable FILENAME contains the name of the current input file.

6.2. Displaying Text

The simplest action is to display (or ***print***) some or all of a record with the awk command print. print copies the input to the output intact. An action without a pattern is executed for all lines. To display each record of the ***sample*** file, use:

```
hostname% awk '{print}' sample
125.1303        krypton loghost
125.0x0733      window
125.1313        core
125.19          haley
hostname%
```

Remember to put single quotes around the awk program as we show here.

More useful than the above example is to print a field or fields from each record. For instance, to display the first two fields in reverse order, type:

```
hostname% awk '{print $2, $1}' sample
krypton 125.1303
window 125.0x0733
core 125.1313
hostname%
```

Items separated by a comma in the print statement are separated by the current output field separator when output. Items not separated by commas are concatenated, so to run the first and second fields together, type:

```
hostname% awk '{print $1 $2}' sample
125.1303krypton
125.0x0733window
125.1313core
125.19haley
hostname%
```

You can use the predefined variables NF and NR; for example, to print each record preceded by the record number and the number of fields, use:

```
hostname% awk '{ print NR, NF, $0 }' sample
1 3 125.1303    krypton loghost
2 2 125.0x0733  window
3 2 125.1313    core
4 2 125.19              haley
hostname%
```

You may divert output to multiple files; the program:

```
hostname% awk '{print $1 >"foo1"; print $2 >"foo2"}' filename
```

writes the first field, $1, on the file *foo1*, and the second field on file *foo2*. You can also use the >> notation; to append the output to the file *foo* for example, say:

```
hostname% awk '{print $1 >>"foo"}' filename
```

In each case, the output files are created if necessary. The filename can be a variable or a field as well as a constant. For example, to use the contents of field 2 as a filename, type:

```
hostname% awk '{print $1 >$2}' filename
hostname%
```

This program prints the contents of field 1 of *filename* on field 2. If you run this on our *sample* file, four new files are created. There is a limit of 10 output files.

Similarly, you can pipe output into another process. For instance, to mail the output of an awk program to *henry*, use:

```
hostname% awk '{ print NR, NF, $0 }' sample | mail henry
```

(See the *Mail User's Guide* in the *Beginner's Guide to the Sun Workstation* for details on mail.)

To change the current output field separator and output record separator, use the variables OFS and ORS. The output record separator is appended to the output of the print statement.

awk also provides the printf statement for output formatting. To format the expressions in the list according to the specification in *format* and print them, use:

```
printf format, expr, expr, ...
```

To print $1 as a floating point number eight digits wide, with two after the decimal point, and $2 as a 10-digit long decimal number, followed by a newline, use:

```
hostname% awk '{printf("%8.2f  %10ld\n",$1,$2)}' filename
```

Notice that you have to specifically insert spaces or tab characters by enclosing them in quoted strings. Otherwise, the output appears all scrunched together. The version of printf is identical to that provided in the C Standard I/O library (see *printf* in *C Library Standard I/O* (3S) in the *System Interface Manual for the Sun Workstation*).

6.3. Specifying Patterns

A pattern in front of an action acts as a selector that determines whether the action is to be executed. You may use a variety of expressions as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary Boolean combinations of these.

6.3.1. BEGIN *and* END

awk has two built-in patterns, BEGIN and END. BEGIN matches the beginning of the input, before the first record is read. The pattern END matches the end of the input, after the last record has been processed. BEGIN and END thus provide a way to gain control before and after processing, for initialization and wrapup.

As an example, the field separator can be set to a colon by:

```
BEGIN   { FS = ":" }
... rest of program ...
```

Or the input lines may be counted by:

```
END  { print NR }
```

If s-2BEGIN is present, it must be the first pattern; s-2END must be the last if used.

6.3.2. Regular Expressions

The simplest regular expression is a literal string of characters enclosed in slashes, like

```
/smith/
```

This is actually a complete awk program which displays all lines which contain any occurrence of the name 'smith'. If a line contains 'smith' as part of a larger word, it is also displayed. Suppose you have a file *testfile* that contains:

```
summertime
smith
blacksmithing
Smithsonian
hammersmith
```

If you use awk on it, the display is:

```
hostname% awk /smith/ testfile
smith
blacksmithing
hammersmith
```

awk regular expressions include the regular expression forms found in the text editor ed and in grep (see the *Commands Reference Manual for the Sun Workstation*). In addition, awk uses parentheses for grouping, | for alternatives, + for 'one or more', and ? for 'zero or one', all as in lex. Character classes may be abbreviated. For example:

```
/[a-zA-Z0-9]/
```

is the set of all letters and digits. As an example, to display all lines which contain any of the names 'Adams,' 'West' or 'Smith,' whether capitalized or not, use:

```
'/[Aa]dams|[Ww]est|[Ss]mith/'
```

Enclose regular expressions (with the extensions listed above) in slashes, just as in **ed** and **sed**. For example:

```
hostname% awk '/[Ss]mith/' testfile
smith
blacksmithing
Smithsonian
hammersmith
```

finds both 'smith' and 'Smith'.

Within a regular expression, blanks and the regular expression metacharacters are significant. To turn off the magic meaning of one of the regular expression characters, precede it with a backslash. An example is the pattern

```
/\/.*\//
```

which matches any string of characters enclosed in slashes.

Use the operators `~` and `!~` to find if any field or variable matches a regular expression (or does not match it). The program

```
$1 ~ /[sS]mith/
```

displays all lines where the first field matches 'smith' or 'Smith.' Notice that this will also match 'blacksmithing', 'Smithsonian', and so on. To restrict it to exactly [sS]mith, use:

```
hostname% awk '$1 - /^[sS]mith$/' testfile
smith
hostname%
```

The caret `^` refers to the beginning of a line or field; the dollar sign `$` refers to the end.

6.3.3. Relational Expressions

An awk pattern can be a relational expression involving the usual relational and arithmetic operators <, <=, ==, !=, >=, and >, the same as those in C. An example is:

```
'$2 > $1 + 100'
```

which selects lines where the second field is at least 100 greater than the first field.

In relational tests, if neither operand is numeric, a string comparison is made; otherwise it is numeric. Thus,

```
hostname% awk '$1 >= "s"' testfile
smith
```

selects lines that begin with an 's', 't', 'u', etc. In the absence of any other information, fields are treated as strings, so the program

```
$1 > $2
```

performs a string comparison between field 1 and field 2.

6.3.4. Combinations of Patterns

A pattern can be any Boolean combination of patterns, using the operators || (or), && (and), and ! (not). For example, to select lines where the first field begins with 's', but is not 'smith', use:

```
hostname% awk '$1 >= "s" && $1 < "t" && $1 != "smith"' testfile
summertime
```

&& and || guarantee that their operands will be evaluated from left to right; evaluation stops as soon as the truth or falsehood is determined.

The program:

```
$1 !=prev {print; prev=$1}
```

displays all lines in which the first field is different from the previous first field.

6.3.5. Pattern Ranges

The pattern that selects an action may also consist of two patterns separated by a comma, as in

```
pattern1, pattern2      { ... }
```

In this case, the action is performed for each line between an occurrence of ***pattern1*** and the next occurrence of ***pattern2*** inclusive. For example, to display all lines between the strings 'sum' and 'black', use:

```
hostname% awk '/sum/, /black/' testfile
summertime
smith
blacksmithing
hostname%
```

while

```
NR == 100, NR == 200 { ... }
```

does the action for lines 100 through 200 of the input.

6.4. Actions

An awk action is a sequence of action statements terminated by newlines or semicolons. These action statements can be used to do a variety of bookkeeping and string manipulating tasks.

6.4.1. Assignments, Variables, and Expressions

The simplest action is an ***assignment***. For example, you can assign 1 to the ***variable x***:

```
x = 1
```

The '1' is a simple expression. awk variables can take on numeric (floating point) or string values according to context. In

```
x = 1
```

x is clearly a number, while in

```
x = "smith"
```

it is clearly a string. Strings are converted to numbers and vice versa whenever context demands it. For instance, to assign 7 to *x*, use:

```
x = "3" + "4"
```

Strings that cannot be interpreted as numbers in a numerical context will generally have numeric value zero, but it is unwise to count on this behavior.

By default, variables other than built-ins are initialized to the null string, which has numerical value zero; this eliminates the need for most BEGIN sections. For example, the sums of the first two fields can be computed by:

```
        { s1 += $1; s2 += $2 }
END     { print s1, s2 }
```

Arithmetic is done internally in floating point. The arithmetic operators are +, −, *, /, and % (mod). For example:

```
NF % 2 == 0
```

displays lines with an even number of fields. To display all lines with an even number of fields, use:

```
NF % 2 == 0
```

The C increment ++ and decrement −− operators are also available, and so are the assignment operators +=, −=, *=, /=, and %=.

An awk pattern can be a ***conditional expression*** as well as a simple expression as in the 'x = 1' assignment above. The operators listed above may all be used in expressions. An awk program with a conditional expression specifies conditional selection based on properties of the individual fields in the record.

6.4.2. Field Variables

Fields in awk share essentially all of the properties of variables — they may be used in arithmetic or string operations, and may be assigned to.

To replace the first field of each line by its logarithm, say:

```
{ $1 = log($1); print }
```

Thus you can replace the first field with a sequence number like this:

```
{ $1 = NR; print }
```

or accumulate two fields into a third, like this:

```
{ $1 = $2 + $3; print $0 }
```

or assign a string to a field:

```
{ if ($3 > 1000)
        $3 = "too big"
  print
}
```

which replaces the third field by 'too big' when it is, and in any case prints the record.

Field references may be numerical expressions, as in

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is considered numeric or string depends on context; fields are treated as strings in ambiguous cases like:

```
if ($1 == $2) ...
```

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields. To split the string 's' into 'array[1]' ..., 'array[n]', use:

```
n = split(s, array, sep)
```

This returns the number of elements found. If the `sep` argument is provided, it is used as the field separator; otherwise `FS` is used as the separator.

6.4.3. String Concatenation

Strings may be concatenated. For example:

```
length($1 $2 $3)
```

returns the length of the first three fields. Or in a `print` statement,

```
print $1 " is " $2
```

prints the two fields separated by ' is '. Variables and numeric expressions may also appear in concatenations.

6.4.4. Built-in Functions

`awk` provides several ***built-in*** functions.

6.4.4.1. `length` *Function*

The `length` function computes the length of a string of characters. This program shows each record, preceded by its length:

```
hostname% awk '{print length, $0}' testfile
10 summertime
5 smith
13 blacksmithing
11 Smithsonian
11 hammersmith
hostname%
```

`length` by itself is a 'pseudo-variable' that yields the length of the current record; `length(argument)` is a function which yields the length of its argument, as in the equivalent:

```
hostname% awk '{print length($0), $0}' testfile
10 summertime
5 smith
13 blacksmithing
11 Smithsonian
11 hammersmith
```

The argument may be any expression.

awk also provides the arithmetic functions sqrt, log, exp, and int, for square root, base *e* logarithm, exponential, and integer part of their respective arguments.

The name of one of these built-in functions, without argument or parentheses, stands for the value of the function on the whole record. The program

```
length < 10 || length > 20
```

displays lines whose length is less than 10 or greater than 20.

6.4.4.2. substr*ing Function*

The function substr(s, m, n) produces the substring of *s* that begins at position *m* (origin 1) and is at most *n* characters long. If *n* is omitted, the substring goes to the end of *s*.

6.4.4.3. index *Function*

The function index(s1, s2) returns the position where the string *s2* occurs in *s1*, or zero if it does not.

6.4.4.4. sprintf *Function*

The function sprintf(*f*, *e1*, *e2*, ...) produces the value of the expressions *e1*, *e2*, and so on, in the printf format specified by *f*. Thus, for example, to set *x* to the string produced by formatting the values of $1 and $2, use:

```
x = sprintf("%8.2f %10ld", $1, $2)
```

6.4.5. Arrays

Array elements are not declared; they spring into existence by being mentioned. Subscripts may have ***any*** non-null value, including non-numeric strings. As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input record to the *NR -th* element of the array *x*. In fact, it is possible in principle though perhaps slow to process the entire input in a random order with the awk program

```
        { x[NR] = $0 }
END     { ... program ... }
```

The first action merely records each input line in the array **x**.

Array elements may be named by non-numeric values, which gives **awk** a capability rather like the associative memory of Snobol tables. Suppose the input contains fields with values like 'apple', 'orange', etc.
Then the program

```
/apple/         { x["apple"]++ }
/orange/        { x["orange"]++ }
END             { print x["apple"], x["orange"] }
```

increments counts for the named array elements, and prints them at the end of the input.

6.4.6. Flow-of-Control Statements

awk provides the basic flow-of-control statements **if-else**, **while**, **for**, and statement grouping with braces, as in C. We showed the **if** statement in the "Field Variables" section without describing it. The condition in parentheses is evaluated; if it is true, the statement following the **if** is done. The **else** part is optional.

The **while** statement is exactly like that of C. For example, to print all input fields one per line,

```
i = 1
while (i <= NF) {
        print $i
        ++i
}
```

The **for** statement is also exactly that of C:

```
for (i = 1; i <= NF; i++)
        print $i
```

does the same job as the **while** statement above.

There is an alternate form of the **for** statement which is suited for accessing the elements of an associative array:

```
for (i in array)
        statement
```

does *statement* with *i* set in turn to each element of *array*. The elements are accessed in an apparently random order. Chaos will ensue if *i* is altered, or if any new elements are accessed during the loop.

The expression in the condition part of an **if**, **while** or **for** can include relational operators like **<**, **<=**, **>**, **>=**, **==** ('is equal to'), and **!=** ('not equal to'); regular expression matches with the match operators **~** and **!~**; the logical operators **||**, **&&**, and **!**; and of course parentheses for grouping.

The **break** statement causes an immediate exit from an enclosing **while** or **for**; the **continue** statement causes the next iteration to begin.

The statement **next** causes **awk** to skip immediately to the next record and begin scanning the patterns from the top. The statement **exit** causes the program to behave as if the end of the input had occurred.

You may put comments in awk programs: begin them with the character # and end them with the end of the line, as in

```
print x, y      # this is a comment
```

Chapter 7

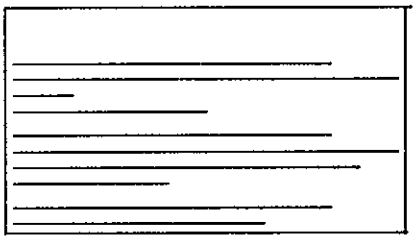
Introduction to Document Preparation

The main document preparation programs in the Sun System are ***nroff*** and ***troff***. These programs handle one or more files containing both the text to be formatted and requests specifying how the output should look. From this input, the programs produce formatted output: ***nroff*** on typewriter-like terminals, and ***troff*** on a phototypesetter. Although they are separate programs, ***nroff*** and ***troff*** are compatible; they share the same command language and produce their output from the same input file. Descriptions here apply to both ***troff*** and ***nroff*** unless indicated otherwise.

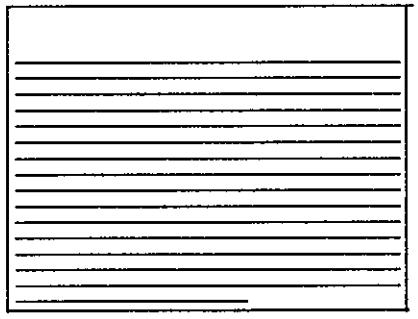
7.1. What Do Text Formatters Do?

You can type in the text of a document on lines of any length, and the text formatters produce lines of uniform length in the finished document. This is called ***filling***, which means that the formatter collects words from what you type in the input file, and places them on an output line until no more will fit within a given line length. It ***hyphenates*** words automatically, so a line may be completed with part of a word to produce the right line length. It also ***adjusts*** a line after it has been filled by inserting spaces between words as necessary to bring the text exactly to the right margin. Examples of filling and adjusting follow:

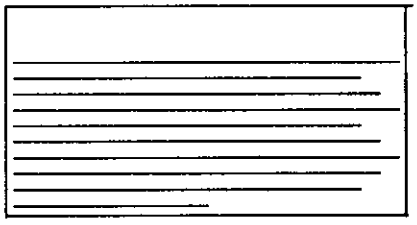
Unfilled text looks like:



Filled and adjusted text looks like:



Filled but not adjusted text looks like:



Given a file of input consisting only of lines of text without any formatting requests, the formatter simply produces a continuous stream of filled, adjusted and hyphenated output.

To obtain paragraphs, numbered sections, multiple column layout, tops and bottoms of pages, and footnotes, for example, require the addition of formatting requests. Requests look like '*.xx*' where *xx* is one or two lower-case letters or a lower-case letter and a digit. Refer to ***Formatting Documents with 'nroff' and 'troff'*** for details.

7.2. What is a Macro Package?

Nroff and ***troff*** provide a flexible, sophisticated command language for requesting operations like those just mentioned. They are very flexible, but this flexibility can make them difficult to use because you have to use several requests to produce a simple format. For this reason, it's a good idea to use a ***macro package.*** A macro is simply a ***predefined sequence of nroff requests or text*** which you can use by including just one request in your input file. You can then handle repetitious tasks, such as starting paragraphs and numbering pages, by typing one macro request each time instead of several. A macro looks like '*.XX*' where *XX* is one or two upper-case letters or an upper-case letter and a digit.

A macro package also does a lot of things without the instructions that you have to give ***nroff***, footnotes and page transitions for example. Some packages set up a page layout style by default, but you can change that style if you wish. Although a macro package offers only a limited subset of the wide range of formatting possibilities that ***nroff*** provides, it is much easier to use. We explain how to use a macro package in conjunction with ***nroff*** and ***troff*** in ***Displaying and Printing Documents.***

Sample input with both formatting requests, macros in this case, and text looks like:

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

Refer to ***Formatting Documents with the −ms Macros*** and to ***Quick References*** in this chapter for more information on macros.

7.3. What is a Preprocessor?

A ***preprocessor*** is a program that you run your text file through first before passing it on to a text formatter. You can put tables in a document by ***preprocessing*** a file with the table-builder called ***tbl.*** You can add mathematical equations with their special fonts and symbols with the equation formatters, ***eqn*** for ***troff*** files and ***neqn*** for ***nroff*** files. These preprocessors convert material entered in their specific command languages to straight ***troff*** or ***nroff*** input. Those text formatters then produce the tables or mathematical equations for the output.

What you type in a file is very much the same as for simple formatting. You include table or equation material in your ***troff*** input file along with ordinary text and add several specific ***tbl*** or ***eqn*** requests. Refer to ***Formatting Tables with 'tbl'*** and ***Formatting Mathematics with 'eqn'*** for details.

7.4. Typesetting Jargon

There are several printer's measurement terms that are borrowed from traditional typesetting. These terms describe the size of the letters, the distance between lines and paragraphs, how long each line is, where the text is placed on the page, and so on.

Point ***Points*** specify the ***size*** of a letter or ***type***. A point measures about 1/72 of an inch, which means that there are 72 points to the inch. This manual is in 10-point type, for instance.

Ems and *Ens*

Ems and ***ens*** are measures of distance and are proportional to the type size being used. An ***em*** is the distance equal to the number of points in the width of the letter 'm' in that point size. For examples, here's an em in several point sizes followed by an em dash to show why this is a ***proportional*** unit of measure. You wouldn't want a 20-point dash if you are printing the rest of a document in 12-point. Here's 12-point:

And here's 20-point:

An ***en*** space is one half of an ***em*** or about the width of the letter 'n'. They are typically used for indicating indention.

Vertical Spacing

Vertical spacing called ***leading*** (pronounced 'led–ing') is the distance between the bottom of one line and the bottom of the next. This manual has 12-point vertical spacing for example. The rule of thumb is that the spacing be 20% bigger than the character size for easy readability. A printer would call the ratio for this manual "Ten over twelve."

Paragraph Depth

As there is a specification for the distance between lines, there is also a term for the space between paragraphs. This is the ***paragraph depth.*** If you are using the standard '.PP' macro, for instance, the paragraph depth is whatever one vertical space has been set to.

Paragraph Indent

This is the amount of space that the first line is indented in relation to the rest of the paragraph. If you use a '.PP' macro to format a standard indented paragraph, the indent is two em spaces as shown by the first line in this paragraph.

Line Length

Line length specifies the width of text on a page. Here we use a 6 1/2-inch line length. Shortening the line length generally makes text easier to read. Recall that many magazines and newspapers have 2-1/4 inch columns for quick reading.

Page Offset

Page offset determines the left margin, that is how far in from the left edge of the paper the text is set. On a normal 8-1/2 by 11 letter-size page, the page offset is normally 26/27 of an inch.

Indent The ***indent*** of text is the distance the text is set in from the page offset. The indent emphasizes the text by setting it off from the rest.

7.5. Hints for Typing in Text

The following provides a few tricks for typing in text and for further online editing and formatting.

- A period (.) or apostrophe (') as the first character on a line indicates that the line contains a formatting request. If you type a line of text beginning with either of these ***control characters***, ***nroff*** tries to interpret them as a request, and the rest of the text on that line disappears. If you ***have*** to print a period or an apostrophe as the first character, escape their normal meanings by prefixing them with a backslash and an ampersand, **\&...**, for instance.
- Following the control character is a one- or two-character ***name*** of a formatting request. As described earlier, ***nroff*** and ***troff*** names usually consist of one or two lower-case letters or a lower-case letter and a digit. Macro package names usually consist of one or two upper-case letters or one upper-case letter and a digit. For example, '.sp' is an ***nroff*** request for a space and '.PP' is an ***−ms*** macro request for an indented paragraph.
- End a line of text with the end of a word along with any trailing punctuation. *Nroff* inserts a space between whatever ends one line of input text and whatever begins the next.
- Start lines in the input file with something other than a space. A space at the beginning of an input line creates a ***break*** at that point in the output and ***nroff*** skips to a new output line, interrupting the process of filling and adjusting. This is the easiest way to get spaces between paragraphs, but it does not leave much flexibility for changing things later.
- Some requests go on a line by themselves, while others can take one or more additional pieces of information on the same line. These extra pieces of information on the request line are called ***arguments***. Separate them from the request name and from each other by one or more spaces. Sometimes the argument is a piece of text on which the request operates; other times it can be some additional information about what the request is to do. For example, the vertical space request '.sp 3' shows an ***nroff*** request with one argument. It requests three blank spaces.

7.6. Types of Paragraphs

There are several types of paragraphs. When should you use one type of paragraph instead of another? Here are a few words about paragraphs, their characteristics, and formatting in general. See the ***Types of Paragraphs*** figure that follows for examples.

Use regular indented and block paragraphs for narrative descriptions. It is a matter of style as to which type you choose to use. In general, indented paragraphs remove the need for extra space between paragraphs — the indent tells you where the start of the new paragraph is. Most business communication is done with block paragraphs.

If you want to indicate a set of points without any specific order, use a bulleted list. For example:

There are many kinds of coffee:

- Jamaica Blue Mountain
- Colombian
- Java
- Mocha
- French Roast
- Major Dickenson's Blend

When you want to describe a set of things in some order, such as a step-by-step procedure, use a numbered list:

To repair television, follow these steps:

1. Remove screws in rear casing.
2. Carefully slide out picture tube.
3. Gently smash with hammer.

Use description lists to explain a set of related or unrelated things, or sometimes to highlight keywords. For instance,

Options

−v	Verbose
−f *filename*	Take script from *filename*
−o	Use old format

In typographic parlance, anything that is not part of the "body text" — regular paragraphs and such — is considered a ***display***, and often has to be specially handled. Generally a display is "displayed" exactly as you type it or draw it originally, with no interference from the formatter. Displays are used to set off important text, special effects, drawings, or examples, as we do throughout this manual, The following paragraph is a ***display***.

Tom appeared on the sidewalk with a bucket of whitewash and
a long-handled brush.
He surveyed the fence, and all gladness left him and
a deep melancholy settled down upon his spirit.
Thirty yards of board fence nine feet high.
Life to him seemed hollow, and
existence but a burden.

Quotations set off quoted material from the rest of the text for emphasis. For example,

"... in the conversation between Alice and the Queen, we read this piece of homespun philosophy:

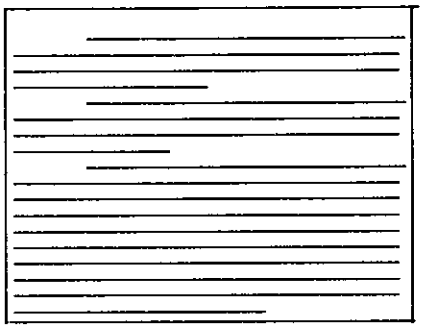
> "A slow sort of country!" said the Queen. "Now, ***here***, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"

Through the Looking Glass
Lewis Carroll

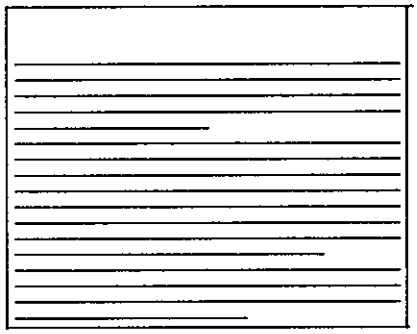
Examine the following thumbnail sketches of paragraph types to see how each can serve a special function:

Table 7-1: Types of Paragraphs

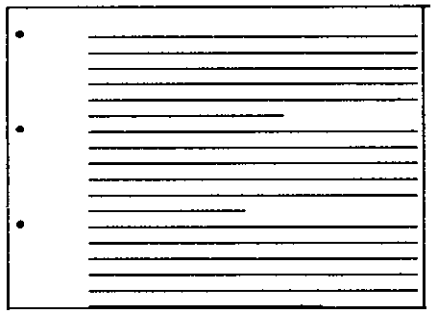
Indented — .PP



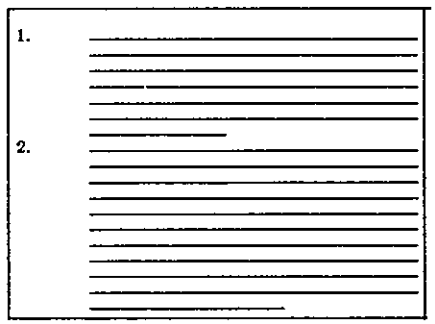
Left Block — .LP



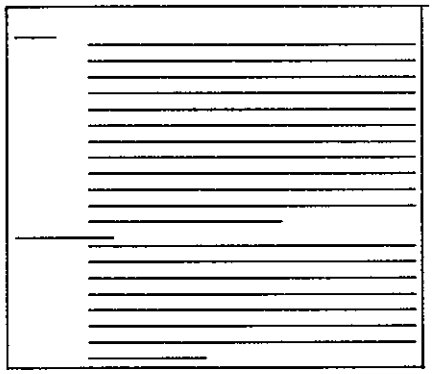
Bulleted — .IP \(bu



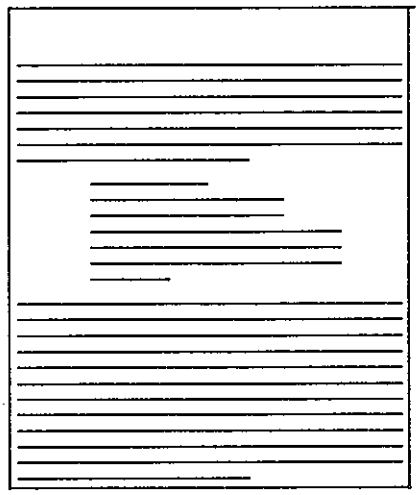
Numbered — .IP 1.



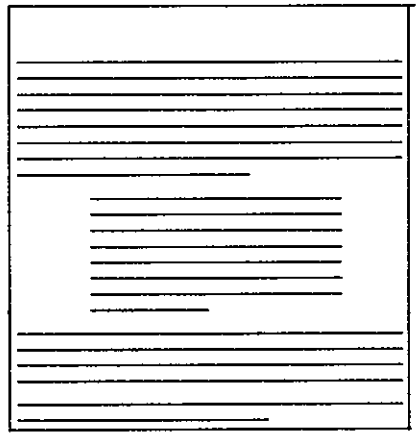
Description Lists — .IP " " *n*



Display — .DS



Quotation — .QP



7.7. Quick References

This section[1] provides some simple templates for producing your documents with the **−ms** macro package. Remember that for a quick, paginated, and justified document, you can simply type an '.LP' to start your document, and then type in the text separated by blank lines to produce paragraphs. Type a space and RETURN to get a blank line.

Throughout the examples, input is shown in

bold Times Roman font

while the output is shown in

this Times Roman font.

7.7.1. Displaying and Printing Documents

Use the following to format and print your documents. You can use either ***nroff*** or ***troff*** depending on the output you desire. Use ***nroff*** to either display formatted output on your workstation screen or to print a formatted document. The default is to display on the standard output, your workstation screen. For easy viewing, pipe your output to ***more*** or redirect the output to a file.

Using ***troff*** or your installation's equivalent prepares your output for phototypesetting.

[1] Some of the material in this section is derived from *A Guide to Preparing Documents with '−ms'*, M.E. Lesk, Bell Laboratories, Murray Hill, New Jersey.

Table 7-2: How to Display and Print Documents

What You Want to Do	How to Do It
Display simple text	**nroff** *—options files*
Display text with tables only	**tbl** *files* ¦ **nroff** *—options*
Display text with equations only	**neqn** *files* ¦ **nroff** *—options*
Display text with both tables and equations	**tbl** *files* ¦ **neqn** ¦ **nroff** *—options*
Print raw text and requests	**pr** *files* ¦ **lpr** *—Pprinter*
Print text	**nroff** *—options files* ¦ **lpr** *—Pprinter*
Print text with tables only	**tbl** *files* ¦ **nroff** *—options* ¦ **lpr** *—Pprinter*
Print text with equations only	**neqn** *files* ¦ **nroff** *—options* ¦ **lpr** *—Pprinter*
Print text with both tables and equations	**tbl** *files* ¦ **neqn** ¦ **nroff** *—options* ¦ **lpr** *—Pprinter*
Phototypeset simple text	**troff** *—options files*
Phototypeset text with tables	**tbl** *files* ¦ **troff** *—options*
Phototypeset text with equations	**eqn** *files* ¦ **troff** *—options*
Phototypeset text with both tables and equations	**tbl** *files* ¦ **eqn** ¦ **troff** *—options*

7.7.2. Technical Memorandum

Here we provide a sample format for a technical memorandum.

Input:

```
.DA March 11, 1983
.TL
An Analysis of
Cucumbers and Pickles
.AU
A. B. Hacker
.AU
C. D. Wizard
.AI
Stanford University
Stanford, California
.AB
This abstract should be short enough to
fit on a single page cover sheet.
It provides a summary of memorandum
contents.
.AE
.NH
Introduction.
.PP
Now the first paragraph of actual text ...
...
Last line of text.
.NH
References
```

Output:

Date: March 11, 1983

An Analysis of
Cucumbers and Pickles

A. B. Hacker
C. D. Wizard

Stanford University
Stanford, California

ABSTRACT

This abstract should be short enough to fit on a single page cover sheet. It provides a summary of memorandum contents.

1. Introduction.

Now the first paragraph of actual text ...

1. References

7.7.3. Section Headings for Documents

.NH	**.SH**
Introduction.	**Appendix I**
.PP	**.PP**
text text text	***text text text***
1. Introduction	Appendix I
text text text	text text text

7.7.4. Changing Fonts

The following table shows the easiest way to change the default roman font to italic or bold. To change the font of a single word, put the word on the same line as the macro. To change more than one word, put them on the line following the macro.

Input	Output
.I Hello	*Hello*
.I **Puts this line in italics.**	*Puts this line in italics.*
.B Goodbye	**Goodbye**
.B **Prints this line in bold.**	**Prints this line in bold.**
.R **Prints this line in roman.**	Puts this line in roman.

7.7.5. Making a Simple List

Use the following template for a simple list.

Input:

```
.IP 1.
J. Pencilpusher and X. Hardwired,
.I
A New Kind of Set Screw,
.R
Proc. IEEE
.B 75
(1976), 23-255.
.IP 2.
H. Nails and R. Irons,
.I
Fasteners for Printed Circuit Boards,
.R
Proc. ASME
.B 23
(1974), 23-24.
.LP  (terminates list)
```

Output:

1. J. Pencilpusher and X. Hardwired, *A New Kind of Set Screw*, Proc. IEEE **75** (1976), 23-255.
2. H. Nails and R. Irons, *Fasteners for Printed Circuit Boards*, Proc. ASME **23** (1974), 23-24.

7.7.6. Multiple Indents for Lists and Outlines

This template shows how to format lists or outlines.

Input:

```
This is ordinary text to point out
the margins of the page.
.IP 1.
First level item
.RS
.IP a)
Second level.
.IP b)
Continued here with another second
level item, but somewhat longer.
.RE
.IP 2.
Return to previous value of the
indenting at this point.
.IP 3.
Another
line.
```

Output:

This is ordinary text to point out the margins of the page.

1. First level item
   a) Second level.
   b) Continued here with another second level item, but somewhat longer.
2. Return to previous value of the indenting at this point.
3. Another line.

7.7.7. Displays

A display does not fill or justify the text. It keeps the text together, and sets the lines off from the rest.

Input:

```
text text text text text text
.DS
and now
for something
completely different
.DE
text text text text text text
```

Output:

hoboken harrison newark roseville avenue grove street east orange brick church orange highland avenue mountain station south orange maplewood millburn short hills summit new providence

and now
for something
completely different

murray hill berkeley heights gillette stirling millington lyons basking ridge bernardsville far hills peapack gladstone

Options: '.DS L': left-adjust; '.DS C': line-by-line center; '.DS B': make block, then center.

7.7.8. Footnotes

For automatically-numbered footnotes, put the predefined string ** at the end of the text you want to footnote like this:[2]

```
you want to footnote like this:\**
.FS
Here's a numbered footnote.
.FE
```

To mark footnotes with other symbols, put the symbol as the first argument to '.FS' and at the end of the text you want to footnote like this:†

```
and at the end of the text you want tot footnote like this:\(dg
.FS \(dg
You can also use an asterisk (*) or a double dagger ‡ (\(dd).
.FE
```

7.7.9. Keeping Text Together — Keeps

Lines bracketed by the following commands are kept together, and will appear entirely on one page:

.KS not moved **.KF** may float

[2] Here's a numbered footnote.

† You can also use an asterisk (*) or a double dagger ‡ (\(dd).

.KE through text **.KE** in text

7.7.10. Double-Column Format

Put a '.2C' at the beginning of the material you want printed in two columns. To return to one column, use '.1C'. Note that '.1C' breaks to a new page.

Input:

```
.TL
The Declaration of Independence
.2C
.PP
When in the course of human events, it becomes necessary for one people to dissolve the political bonds which have connected them with another, and to assume among the powers of the earth the separate and equal station to which the laws of Nature and of Nature's God entitle them, a decent respect to the opinions of ...
```

7.7.11. Sample Tables

Two sample table templates follow.

Input:

```
.TS
box center tab (/);
lB lB
l l.
Column Header     Column Header
_
text/text
text/text
text/text
text/text
.TE
```

Output:

Column Header	Column Header
text	text
text	text
text	text
text	text

Input:

```
.TS
allbox tab (/);
c s s
c c c
n n n.
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*
.TE
* (first quarter only)
```

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)

The meanings of the key-letters describing the alignment of each entry are:

Letter	Meaning	Letter	Meaning
c	center	n	numerical
r	right-adjust	a	subcolumn
l	left-adjust	s	spanned

The global table options are center, expand, box, doublebox, allbox, tab (*x*) and linesize (*n*).

Input:

```
.TS
box, center tab(/);
c c
l l.
Name/Definition
.sp
Gamma/$GAMMA (z) = int sub 0 sup inf t sup {z-1} e sup -t dt$
Sine/$sin (x) = 1 over 2i ( e sup ix - e sup -ix )$
Error/$ roman erf (z) = 2 over sqrt pi int sub 0 sup z e sup {-t sup 2} dt$
Bessel/$ J sub 0 (z) = 1 over pi int sub 0 sup pi cos ( z sin theta ) d theta $
Zeta/$ zeta (s) = sum from k=1 to inf k sup -s ~~( Re~s > 1)$
.TE
```

Output:

Name	Definition
Gamma	$\Gamma(z)=\int_0^\infty t^{z-1}e^{-t}\,dt$
Sine	$\sin(z)=\frac{1}{2i}(e^{iz}-e^{-iz})$
Error	$\mathrm{erf}(z)=\frac{2}{\sqrt{\pi}}\int_0^z e^{-t^2}dt$
Bessel	$J_0(z)=\frac{1}{\pi}\int_0^\pi \cos(z\sin\theta)\,d\theta$
Zeta	$\zeta(s)=\sum_{k=1}^{\infty}k^{-s}$ (Re $s>1$)

7.7.12. Writing Mathematical Equations

A displayed equation is marked with an equation number at the right margin by adding an argument to the '.EQ' line:

Input:

```
.EQ (1.3)
x sup 2 over a sup 2 ~=~ sqrt {p z sup 2 +qz+r}
.EN
```

A displayed equation is marked with an equation number at the right margin by adding an argument to the EQ line:

Output:

$$\frac{x^2}{a^2} = \sqrt{pz^2+qz+r} \qquad (1.3)$$

Input:

```
.EQ I (2.2a)
bold V bar sub nu~=~left [ pile {a above b above
c } right ] + left [ matrix { col { A(11) above .
above . } col { . above . above .} col {. above .
above A(33) }} right ] cdot left [ pile { alpha
above beta above  gamma } right ]
.EN
```

Output:

$$\nabla_{\nu} = \begin{bmatrix} a \\ b \\ c \end{bmatrix} + \begin{bmatrix} A(11) & . & . \\ . & . & . \\ . & . & A(33) \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \qquad (2.2a)$$

Input:

```
.EQ L
F hat ( chi ) ~ mark = ~ ¦ del V ¦ sup 2
.EN
.EQ L
lineup =~ {left ( {partial V} over {partial x} right ) } sup 2 + { left ( {partial
V} over {partial y} right ) } sup 2 ~~~~~~ lambda -> inf
.EN
```

Output:

$$F(\chi) = |\nabla V|^2$$

$$= \left(\frac{\partial V}{\partial x}\right)^2 + \left(\frac{\partial V}{\partial y}\right)^2 \qquad \lambda \rightarrow \infty$$

Input:

```
$ a dot $,  $ b dotdot$,  $ xi tilde times y vec$.
```

Output:

$\dot{a}$, $\tilde{b}$, $\xi \times \vec{y}$.

(with delim $$ on).

7.7.13. Registers You Can Change

Table 7-3: Registers You Can Change

Controls	Register	Controls	Register
Line length	.nr LL 7i	Title length	.nr LT 7i
Point size	.nr PS 9	Vertical spacing	.nr VS 11
Column width	.nr CW 3i	Intercolumn spacing	.nr GW .5i
Margins — head and foot	.nr HM .75i .nr FM .75i	Paragraph indent	.nr PI 2n
Paragraph spacing	.nr PD 0	Page offset	.nr PO 0.5i
Page heading	.ds CH Appendix (center) ds RH 7-25-76 (right) .ds LH Private (left)	Page footer	.ds CF Draft .ds LF .ds RF similar
Page numbers	.nr % 3		

Chapter 8

Formatting Documents with the —ms Macros

This chapter[3] describes the new **—ms** macro package for preparing documents with `nroff` and `troff` on the Sun system. The **—ms** Request Summary at the end of this chapter provides a quick reference for all the **—ms** macros and for useful displaying and printing commands. If you are acquainted with **—ms**, there is a quick reference for the *new* requests and string definitions as well. The differences between the new and the old **—ms** macro packages are described in the section entitled "Changes in the New **—ms** Macro Package". Displaying and Printing Documents with **—ms** describes how you can produce documents on either your workstation, printer, or phototypesetter without changing the text and formatting request input.

8.1. Changes in the New —ms Macro Package

The old **—ms** macro package has been revised, and the new macro package assumes the name **—ms**. There are some extensions to previous **—ms** macros and a number of new macros, but all the previously documented **—ms** macros still work exactly as they did before, and have the same names as before. The new **—ms** macro package includes several bug fixes, including a problem with the single-column `.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` bibliographic citation macros. Macros used only at Bell Laboratories have been removed from the new version. We list them at the end of this chapter in "**—ms** Request Summary".

8.2. Displaying and Printing Documents with —ms

After you have prepared your document with text and **—ms** formatting requests and stored it in a file, you can display it on your workstation screen or print it with `nroff` or `troff` with the **—ms** option to use the **—ms** macro package. A good way to start is to pipe your file through `more` for viewing:

```
hostname% nroff -ms filename ... | more
```

If you forget the **—ms** option, you get continuous, justified, unpaginated output in which **—ms** requests are ignored. You can format more than one file on the command line at a time, in which case `nroff` simply processes all of them in the order they appear, as if they were one file. There are other ***options*** to use with `nroff` and `troff`; see the ***Commands Reference Manual for the Sun Workstation*** for details.

---

[3] The material in this chapter is derived from *A Revised Version of '—ms', B. Tuthill, University of California, Berkeley; Typing Documents on the UNIX System: Using the '—ms' Macros with 'troff' and 'nroff'*, M.E. Lesk, Bell Laboratories, Murray Hill, New Jersey; and *Document Formatting on UNIX: Using the '—ms' Macros*, Joel Kies, University of California, Berkeley.

You can get preview and final output of various sorts with the following commands. To send `nroff` output to the line printer, type:

```
hostname% nroff -ms filename | lpr -RIprinter
```

To produce a file with tables, use:

```
hostname% tbl filename | nroff -ms | lpr -printer
```

To produce a file with equations, type:

```
hostname% neqn filename | nroff -ms | lpr -printer
```

To produce a file with tables and equations, use the following order:

```
hostname% tbl filename | neqn | nroff -ms | lpr -printer
```

To print your document with `troff`, use:

```
hostname% troff -ms filename | lpr -t -printer
```

See `lpr` in the Commands Reference Manual for the Sun Workstation for details on printing.

8.3. What Can Macros Do?

Macros can help you produce paragraphs, lists, sections (optionally with automatic numbering), page titles, footnotes, equations, tables, two-column format, a table of contents, endnotes, running heads and feet, and cover pages for papers. As with other formatting utilities such as `nroff` and `troff`, you prepare text interspersed with formatting requests. However, the macro package, which itself is written in `troff` commands, provides higher-level commands than those provided with the basic `troff` program. In other words, you can do a lot more with just one macro than with one `troff` request.

8.4. Formatting Requests

An —**ms** request usually consists of one or two upper-case characters, and usually in the form *.XX*.

The easiest way to produce simple formatted text is to put an `.LP` request at the start of the document and add your text, leaving just a blank line to separate paragraphs. The `.LP` request produces a left-blocked paragraph, as we used throughout this chapter. Your output will have paragraphs and be paginated with right and left-justified margins.

When you use a macro package, you type in text as you normally do and intersperse it with formatting ***requests***. For example, instead of spacing in with the space bar or typing a tab to indent for paragraphs, type a line with the `.PP` request before each paragraph. When formatted, this leaves a space and indents the first line of the following paragraph.

Note: You cannot just begin a document with a line of text. You must include some —**ms** request before any text input. When in doubt, use `.LP` to properly ***initialize*** the file, although any of the requests `.PP`, `.LP`, `.TL`, `.SH`, `.NH` is good enough. See the section "Cover Sheets and Title Pages" later in this chapter for the correct arrangement of requests at the start of a document.

8.4.1. Paragraphs

You can produce several different kinds of paragraphs with the **—ms** macro package: standard, left-block, indented, labeled, and quoted.

8.4.1.1. Standard Paragraph — .PP

To get an ordinary paragraph, use the .PP request, followed on subsequent lines by the text of the paragraph. For example, you type:

```
.PP
Tom appeared on the sidewalk with a bucket of whitewash and a long-handled
brush.
He surveyed the fence, and all gladness left him and a deep melancholy
settled down upon his spirit.
Thirty yards of board fence nine feet high.
Life to him seemed hollow, and
existence but a burden.
```

to produce:

Tom appeared on the sidewalk with a bucket of whitewash and a long-handled brush. He surveyed the fence, and all gladness left him and a deep melancholy settled down upon his spirit. Thirty yards of board fence nine feet high. Life to him seemed hollow, and existence but a burden.

8.4.1.2. Left-Block Paragraph — .LP

You can also produce a left-block paragraph, like those in this manual, with .LP. The first line is not indented as it is with the .PP request. For example, you type:

```
.LP
Tom appeared ...
```

to produce:

Tom appeared on the sidewalk with a bucket of whitewash and a long-handled brush. He surveyed the fence, and all gladness left him and a deep melancholy settled down upon his spirit. Thirty yards of board fence nine feet high. Life to him seemed hollow, and existence but a burden.

There are default values for the vertical spacing before paragraphs and for the width of the indentation. To change the paragraph spacing, see the section "Modifying Default Features".

8.4.1.3. Indented Paragraph — .IP

Another kind of paragraph is the indented paragraph, produced by the .IP request. These paragraphs can have hanging numbers or labels. For example:

```
.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, ...
.LP
```

produces

[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. Here we used the .LP request.

More sophisticated uses of .IP are also possible. If the label is omitted, for example, you get a plain block indent:

```
Tom appeared on the sidewalk with a bucket of whitewash and a long-handled
brush.
.IP
He surveyed the fence, and all gladness left him and a deep melancholy
settled down upon his spirit.
Thirty yards of board fence nine feet high.
Life to him seemed hollow, and
existence but a burden.
.LP
```

which produces

Tom appeared on the sidewalk with a bucket of whitewash and a long-handled brush.

> He surveyed the fence, and all gladness left him and a deep melancholy settled down upon his spirit. Thirty yards of board fence nine feet high. Life to him seemed hollow, and existence but a burden.

If a non-standard amount of indenting is required, specify it after the label in character positions. It remains in effect until the next .PP or .LP. Thus, the general form of the .IP request contains two additional fields: the label and the indenting length. For example,

```
.IP "Example one:" 15
Notice the longer label, requiring larger
indenting for these paragraphs.
.IP "Example two:"
And so forth.
.LP
```

produces this:

Example one: Notice the longer label, requiring larger indenting for these paragraphs.

Example two: And so forth.

Notice that you must enclose the label in double quote marks because it contains a space; otherwise, the space signifies the end of the argument. The indentation request above is in the number of ***ens***, a unit of dimension used in typesetting. An ***en*** is approximately the width of a lowercase 'n' in the particular point size you are using.

The .IP macro adjusts properly by causing a break to the next line if you type in a label longer than the space you allowed for. For example, if you have a very long label and have allowed 10 n spaces for it, your input looks like:

```
.IP "A very, very, long and verbose label" 10
And now here's the text that you want.
And now here's the text that you want.
And now here's the text that you want.
And now here's the text that you want.
And now here's the text that you want.
```

And your output is adjusted accordingly with a break between the label and the text body:

A very, very, long and verbose label
And now here's the text that you want. And now here's the text that you want. And now here's the text that you want. And now here's the text that you want. And now here's the text that you want.

8.4.1.4. Nested Indentation — .RS *and* .RE

It is also possible to produce multiple (or ***relative***) nested indents; the .RS request indicates that the next .IP starts its indentation from the current indentation level. Each .RE undoes one level of indenting, so you should balance .RS and .RE requests. Think of the .RS request as 'move right' and the .RE request as 'move left'. As an example:

```
.IP I.
South Bay Area Restaurants
.RS
.IP A.
Palo Alto
.RS
.IP 1.
La Terrasse
.RE
.IP B.
Mountain View
.RS
.IP 1.
Grand China
.RE
.IP C.
Menlo Park
.RS
.IP 1.
Late for the Train
.IP 2.
Flea Street Cafe
.RE
.RE
.LP
```

results in

I. South Bay Area Restaurants

A. Palo Alto

1. La Terrasse

B. Mountain View

1. Grand China

C. Menlo Park

1. Late for the Train
2. Flea Street Cafe

Note the two .RE requests in a row at the end of the list. Remember that you need one ***end*** for each ***start***.

8.4.1.5. Quoted Paragraph — .QP

All of the variations on .LP leave the right margin untouched. Sometimes, you need a a paragraph indented on both right and left sides. To set off a quotation as such, use:

```
.QP
Precede each paragraph that you want offset as a quotation
with a .QP. This produces a paragraph like this.
Notice that the right edge is also indented from the right margin.
```

to produce

> Precede each paragraph that you want offset as a quotation with a .QP. This produces a paragraph like this. Notice that the right edge is also indented from the right margin.

8.4.2. Section Headings — .SH *and* .NH

There are two varieties of section headings, unnumbered with .SH and numbered with .NH. In either case, type the text of the section heading on one or more lines following the request. End the section heading by typing a subsequent paragraph request or another section heading request. When printed, one line of vertical space precedes the heading, which begins at the left margin. `nroff` offsets the heading with blank lines, while `troff` sets it in **boldface** type. .NH section headings are numbered automatically. The macro takes an argument number representing the ***level-number*** of the heading, up to 5. A third-level section number is one like '1.2.1'. The macro adds one to the section number at the requested level, as shown in the following example:

```
.NH
Bay Area Recreation
.NH 2
Beaches
.NH 3
San Gregorio
.NH 3
Half Moon Bay
.NH 2
Parks
.NH 3
Wunderlich
.NH 3
Los Trancos
.NH 2
Amusement Parks
.NH 3
Marine World/Africa USA
```

generates:

2. Bay Area Recreation

2.1 Beaches

2.1.1 San Gregorio

2.1.2 Half Moon Bay

2.2 Parks

2.2.1 Wunderlich

2.2.2 Los Trancos

2.3 Amusement Parks

2.3.1 Marine World/Africa USA

.NH without a level-number means the same thing as .NH 1, and .NH 0 cancels the numbering sequence in effect and produces a section heading numbered 1.

8.4.3. Cover Sheets and Title Pages

—**ms** provides a group of macros to format items that typically appear on the cover sheet or title page of a formally laid-out paper. You can use them selectively, but if you use several, you must put them in the order shown below, normally at or near the beginning of the input file.

The first line of a document signals the general format of the first page. In particular, if it is .RP (released paper), a cover sheet with title and abstract is prepared. The default format is useful for scanning drafts.

Sample input is:

```
.RP                    (Optional; use for released paper format)
.TL
Title of document (one or more lines)
.AU
Author(s)  (may also be several lines)
.AI
Author's institution(s)
.AB
Abstract; to be placed on the cover sheet of a paper.
Line length is 5/6 of normal; use .ll here to change.
.AE                    (abstract end)
text ...     (begins with .PP)
```

(See *Order of Requests in Input* for a quick example of this scheme.)

If the .RP request precedes .TL, the title, author, and abstract material are printed separately on a cover sheet. The title and author information (not the abstract) is then repeated automatically on page one (the title page) of the paper, without your having to type it again. If you do not include an .RP request, all of this material appears on page one, followed on the same page by the main text of the paper.

To omit some of the standard headings (such as no abstract, or no author's institution), just omit the corresponding fields and command lines. To suppress the word ABSTRACT type .AB no rather than .AB. You can intersperse several .AU and .AI lines to format for multiple authors.

These macros are optional; you may begin a paper simply with a section heading or paragraph request. When you do precede the main text with cover sheet and title page material, include a paragraph or section heading between the last title page request and the beginning of the main text. Don't forget that some —**ms** request must precede any input text.

8.4.4. Running Heads and Feet — LH, CH, RH

The —**ms** macros, by default, print a page heading containing a page number (if greater than 1). You can make minor adjustments to the page headings and 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 `nroff` output, there are two default values: CH is the current page number surrounded on either side by hyphens, and CF contains the current date as supplied by the computer. For `troff` CH also contains the page number, but CF is empty. The other four registers are empty by default for both `nroff` and `troff`. You can use the `.ds` request to assign a value to a string register. For example:

```
.ds RF Draft Only \(em Do Not Distribute
```

This prints the character string

Draft Only — Do Not Distribute

at the bottom right of every page. You do not need to enclose the string in double quote marks. To remove the contents of a string register, simply redefine it as empty. For instance, to clear string register CH, and make the center header blank on the following pages, use the request:

```
.ds CH
```

To put the page number in the right header, use:

```
.ds RH %
```

In a string definition, '%' is a special symbol referring to `nroff`'s automatic page counter. If you want hyphens on either side of the page number, place them on either side of the '%' in the command, that is:

```
.ds RH -%-
```

Remember that putting the page number in the right header as shown above does not remove it from the default CF; you still have to clear out CF.

If you want requests that set the values of string and number registers to take effect on the first page of output, put them at or near the beginning of the input file, before the initializing macro, which in turn must precede the first line of text. Among other functions, the initializing macro causes a 'pseudo page break' onto page one of the paper, including the top-of-page processing for that page. Be sure to put requests that change the value of the PO (page offset), HM (top or head margin), and FM (bottom or foot margin) number registers and the page header string registers before the transition onto the page where they are to take effect.

For more complex formats, you can redefine the macros PT (page top) and BT (page bottom), 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. If you redefine these macros, be careful not to change parameters such as point size or font without resetting them to default values.

8.4.5. Custom Headers and Footers — .OH, .EH, .OF, *and* .EF

You can also 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 the `nroff .tl`, that is, there are three fields (left, center and right), each separated by a single apostrophe. For example, to get odd-page headers with the chapter name followed by the page number and the reverse on even pages, use:

```
.OH 'For Whom the Bell Tolls''Page %'
.EH 'Page %''For Whom the Bell Tolls'
```

Note that it is an error to have an apostrophe in the header text; if you need an apostrophe, use a backslash and apostrophe (\') or a delimiter other than apostrophe 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`.

You can use the `.P1` (P one) macro to print the header on page 1. If you want roman numeral page numbering, use an `.af PN i` request.

8.4.6. Multi-column Formats — .2C and .MC

If you place the request `.2C` in your document, the document will be printed in double column format beginning at that point. This is often desirable on the typesetter. Each column will have a width 7/15 that of the text line length in single-column format, and a gutter (the space between the columns) of 1/15 of the full line length. Remember that when you use the two-column `.2C` request, either pipe the ***nroff*** output through ***col*** or make the first line of the input '.pi /usr/bin/col.'

The .2C request is actually a special case of the .MC request that produces formats of more than two spaces:

.MC [*column width* [*gutter width*]]

This formats output in as many columns of ***column width*** as will fit across the page with a gap of ***gutter width***. You can specify the column width in any unit of scale, but if you do not specify a unit, the setting defaults to ens. .MC without any column width is the same thing as .2C. For example:

```
.MC
Tom appeared on the sidewalk with a bucket of whitewash and a long-handled
brush.
He surveyed the fence, and all gladness left him and a deep melancholy
settled down upon his spirit.
```

To return to single-column output, use .1C. Switching from double to single-column always causes a skip to a new page.

8.4.7. Footnotes — .FS and .FE

Material placed between lines with the commands .FS (footnote) and .FE (footnote end) is collected, remembered, and placed at the bottom of the current page.* The formatting of the footnote is:

```
.FS
* Like this.
.FE
```

By default, footnotes are 11/12th the length of normal text, but you can modify this by changing the FL register (see the "Modifying Default Features" section). When typeset, footnotes appear in smaller size type.

Because the macros only save a passage of text for printing at the bottom of the page, you have to mark the footnote reference in some way, both in the text preceding the footnote and again as part of the footnote text. We use a simple asterisk, but you can use anything you want.

You can also produce automatically-numbered footnotes. Footnote numbers are printed by a pre-defined string (**), which you invoke separately from .FS and .FE. Each time this string is used, it increases the footnote number by one, whether or not you use .FS and .FE in your text. Footnote numbers are superscripted on the phototypesetter and on daisy-wheel terminals, but on low-resolution devices (such as the lpr and a crt), they are bracketed. If you use ** to indicate numbered footnotes, the .FS macro automatically includes the footnote number at the bottom of the page.

This footnote, for example, was produced as follows:[4]

```
This footnote, for example, was produced as follows:\**
.FS
        ...
.FE
```

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: †

```
then give that mark as the first argument to .FS: \(dg
.FS  \(dg
        ...
.FE
```

Footnote numbering is temporarily suspended, because the '**' string is not used. Instead of a dagger, you could use an asterisk * or double dagger ‡, represented as '\(dd'.

---

[3] * Like this.

[4] If you never use the '**' string, no footnote numbers will appear anywhere in the text, including down here. The output footnotes will look exactly like footnotes produced with —mos, the old —ms macro package.

† In the footnote, the dagger will appear where the footnote number would otherwise appear, as shown here.

8.4.8. Endnotes

If you want 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 the .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 gives 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.

8.4.9. Displays and Tables — .DS and .DE

To prepare displays of lines, such as tables, in which the lines should not be re-arranged or broken between pages, enclose them in the requests .DS and .DE:

```
.DS
lines, like the
examples here, are placed
between .DS and .DE
.DE
```

which produces:

lines, like the
examples here, are placed
between .DS and .DE

By default, lines between .DS and .DE are indented from the left margin.

If you don't want the indentation, use .DS L to begin and .DE to produce a left-justified display:

to get
something like
this

You can also center lines with the .DS C and .DE requests:

This is an
example
of a centered display.

Note that each line is centered individually.

A plain .DS is equivalent to .DS I, which indents and left-adjusts. An extra argument to the .DS I or .DS request is taken as an amount to indent. For example, .DS I 3 or .DS 3 begins a display to be indented 3 ens from the margin.

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, and .BD in place of the requests .DS C, .DS L, and .DS B respectively. Use .ID for either a plain .DS or .DS I. You can also specify the amount of indention with the .ID macro.

Use the following table as a quick reference:

Table 8-1: Display Macros

Macro with Keep	Macro without Keep
.DS I	.ID
.DS L	.LD
.DS C	.CD
.DS B	.BD
.DS	.ID

Note: It is tempting to assume that `.DS R` will right adjust lines, but it doesn't work.

8.4.10. Keeping Text Together — `.KS`, `.KE` *and* `.KF`

If you wish to keep a table or other block of lines together on a page, there are 'keep - release' requests. If a block of lines preceded by `.KS` and followed by `.KE` does not fit on the remainder of the current page, it will begin on a new page. There is also a 'keep floating' request. 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 to the top of the next page. `nroff` fills in the current page with the ordinary text that follows the keep in the input file to avoid leaving blank space at the bottom of the page preceding the keep. Thus, no large blank space will be introduced in the document.

In multi-column output, the keep macros attempt to place all the kept material in the same column.

If the material enclosed in a keep requires more than one page, or more than a column in multi-column format, it will start on a new page or column and simply run over onto the following page or column.

8.4.11. Boxing Words or Lines — `.BX` *and* `.B1` *and* `.B2`

To draw rectangular boxes around words, use the request

```
.BX word
```

to print word as shown.

You can box longer pieces of text by enclosing them with `.B1` and `.B2`:

```
.B1
Tom appeared on the sidewalk with a bucket of whitewash and a long-handled
brush.
He surveyed the fence, and all gladness left him and a deep melancholy
settled down upon his spirit.
Thirty yards of board fence nine feet high.
Life to him seemed hollow, and
existence but a burden.
.B2
```

This produces:

Tom appeared on the sidewalk with a bucket of whitewash and a long-handled brush. He surveyed the fence, and all gladness left him and a deep melancholy settled down upon his spirit. Thirty yards of board fence nine feet high. Life to him seemed hollow, and existence but a burden.

8.4.12. Changing Fonts — .I, .B, .R *and* .UL

To get italics on the typesetter or reverse display on the workstation, say:

```
.I
as much text as you want
can be typed here
.R
```

as was done for ***these three words***. The .R request restores the normal (usually Roman) font. If only one word is to be italicized, you can put it on the line with the .I request:

```
.I word
```

and in this case you do not need to use an .R to restore the previous font.

You can print **boldface** font by

```
.B
Text to be set in boldface
goes here
.R
```

As with .I, you can place a single word in boldface font by putting it on the same line as the .B request. Also, when .I or .B is used with a word as an argument, it can take as a second argument any trailing punctuation to be printed immediately after the word but set in normal typeface. For example:

```
.B word )
```

prints

word)

that is, the word in boldface and the closing parenthesis in normal Roman directly adjacent to the word.

If you want actual underlining as opposed to italicizing on the typesetter, use the request

```
.UL word
```

to underline a word. There is no way to underline multiple words on the typesetter.

8.4.13. Changing the Type Size — .LG, .SM *and* .NL

You can specify a few size changes in `troff` output with the requests .LG (make larger), .SM (make smaller), and .NL (return to normal size). The size change is two points (see the "Dimensions" section for a discussion of point size); you can repeat the requests for increased effect (here one .NL canceled two .SM requests). These requests are primarily useful for temporary size changes for a small number of words. They do not affect vertical spacing of lines of text. See the section on "Modifying Default Features" for other techniques for changing the type size and vertical spacing of longer passages.

8.4.14. Dates — .DA and .ND

When you use **—ms**, `nroff` prints the date at the bottom of each page, but `troff` does not. Both `nroff` and `troff` print it on the cover sheet if you have requested one with `.RP`. To make `troff` print the date as the center page footer, say `.DA` (date). To suppress the date, say `.ND` (no date). To lie about the date, type `.DA July 4, 1776`, which puts the specified date at the bottom of each page. The request:

```
.ND September 16, 1959
```

in `.RP` format places the specified date on the cover sheet and nowhere else. Place either `.ND` or `.DA` before the `.RP` Notice this is one instance that you do not need to put double quote marks around the arguments.

8.4.15. Thesis Format Mode — .TM

To format a paper as a thesis, use the `.TM` macro (thesis mode). It is much like the `.th` macro in the **—me** macro package. It puts page numbers in the upper right-hand corner, numbers the first page, suppresses the date, and doublespaces everything except quotes, displays, and keeps. Use it at the top of each file making up your thesis. Calling `.TM` defines the `.CT` macro for chapter titles, which skips to a new page and moves the page number to the center footer. You can use the `.P1` (P one) macro even without thesis mode to print the header on page one, which is suppressed except in thesis mode. If you want roman numeral page numbering, use an `.af PN i` request.

8.4.16. Bibliography — .XP

To format bibliography entries, use the `.XP` macro, which stands for ***exdented paragraph***. It exdents the first line of the paragraph by \n(PI units, usually 5n, the same as the indent for the first line of a `.PP`. An example of exdented paragraphs is:

```
.XP
Lumley, Lyle S., \fISex in Crustaceans: Shell Fish Habits,\fP\^
Harbinger Press, Tampa Bay and San Diego, October 1979.
243 pages.
The pioneering work in this field.
.XP
Leffadinger, Harry A., ``Mollusk Mating Season: 52 Weeks, or All Year?''
in \fIActa Biologica,\fP\^ vol. 42, no. 11, November 1980.
A provocative thesis, but the conclusions are wrong.
```

which produces:

Lumley, Lyle S., ***Sex in Crustaceans: Shell Fish Habits,*** Harbinger Press, Tampa Bay and San Diego, October 1979. 243 pages. The pioneering work in this field.

Leffadinger, Harry A., "Mollusk Mating Season: 52 Weeks, or All Year?" in ***Acta Biologica,*** vol. 42, no. 11, November 1980. A provocative thesis, but the conclusions are wrong.

You do have to italicize the book and journal titles and quote the title of the journal article. You can change the indentation and exdentation by setting the value of number register PI.

8.4.17. Table of Contents — .XS, .XE, .XA, .PX

There are four macros that produce a table of contents. Enclose table of contents entries 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. A sample of typical input and output text is:

```
.XS  ii
Introduction
.XA  1
Chapter 1: Review of the Literature
.XA  23
Chapter 2: Experimental Evidence
.XE
.PX
```

Table of Contents

Introduction ii
Chapter 1: Review of the Literature 1
Chapter 2: Experimental Evidence 23

You can also use the .XS and .XE pairs 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 make the table of contents after finishing your document. To print out the table of contents, use the .PX macro or nothing will happen.

8.4.18. Defining Quotation Marks

To produce quotation marks and dashes that format correctly with both `nroff` and `troff`, there are some string definitions for each of the formatting programs. The *- string yields two hyphens in `nroff`, and produces an em dash — like this one in `troff`. The *Q and *U strings produce " and " in `troff`, but " in `nroff`.

8.4.19. Accent Marks

To simplify typing certain foreign words, the —**ms** macro package defines strings representing common accent marks. There are a large number of optional foreign accent marks defined by the —**ms** macros. All the accent marks available in —**mos** are present, and they all work just as they always did.

For the old accent marks, type the string ***before*** the letter over which the mark is to appear. For example, to print 'téléphone with the old macros, you type:

```
t\*'el\*'ephone
```

Unlike the old accent marks, the new accent strings should be placed ***after*** the letter being accented. Place .AM (accent mark) at the beginning of your document, and type the accent strings ***after*** the letter being accented. A list of both sets of diacritical marks and examples of what they look like follows. *Note:* Do not use the `tbl` macros .TS and .TE with any of the

accent marks as the marks do not line up correctly.

Table 8-2: Old Accent Marks

Accent Name	Input	Output
acute	*'e	é
grave	*`e	è
umlaut	*:u	ü
circumflex	*^e	ê
tilde	*~a	ã
hacek	*Cr	ř
cedilla	*,c	ç

Table 8-3: Accent Marks

Accent Name	Input	Output
acute	e*'	é
grave	e*`	è
circumflex	o*^	ô
cedilla	c*,	ç
tilde	n*~	ñ
question	*?	¿
exclamation	*!	¡
umlaut	u*:	ü
digraphes	*8	ß
hacek	c*v	č
macron	a*_	ā
o-slash	o*/	ø
yogh	kni*3t	kniȝt
angstrom	a*o	å
Thorn	*(Th	Þ
thorn	*(th	þ
Eth	*(D-	Ð
eth	*(d-	ð
hooked o	*q	ǫ
ae ligature	*(ae	æ
AE ligature	*(Ae	Æ
oe ligature	*(oe	œ
OE ligature	*(Oe	Œ

If you want to use these new diacritical marks, don't forget the .AM at the top of your file. Without it, some of these marks will not print at all, and others will be placed on the wrong letter.

8.5. Modifying Default Features

The **—ms** macro package supplies a standard page layout style. The text line has a default length of six inches; the indentation of the first line of a paragraph is five ens; the page number is printed at the top center of every page after page one; and so on for standard papers. You can alter many of these default features by changing the values that control them.

The computer memory locations where these values are stored are called ***number registers*** and ***string registers***. Number and string registers have names like those of requests, one or two characters long. For instance, the value of the line length is stored in a number register named LL. Unless you give a request to change the value stored in register LL, it will contain the standard or default value assigned to it by **—ms**. The "Summary of **—ms** Number Registers table lists the number registers you can change along with their default values.

8.5.1. Dimensions

To change a dimension like the line length from its default value, reset the associated number register with the `troff` request `.nr` (number register):

```
.nr LL 5i
```

The first argument, `LL`, is the name of a number register, and the second, `5i` is the value being assigned to it. In the case above, the line length is adjusted from the default six inches to five inches. As another example, consider:

```
.nr PS 9
```

which makes the default point size 9 point.

The value may be expressed as an integer or may contain a decimal fraction. When setting the value of a number register, it is almost always necessary to include a unit of scale immediately after the value. In the example above, the 'i' as the unit of scale lets `troff` know you mean five inches and not five of some other unit of distance. But the point size (PS) and vertical spacing (VS) registers are exceptions to this rule; ordinarily they should be assigned a value as a number of points ***without indicating the unit of scale.*** For example, to set the vertical spacing to 24 points, or one-third of an inch (double-spacing), use the request:

```
.nr VS 24
```

In the unusual case where you want to set the vertical spacing to more than half an inch (more than 36 points), include a unit of scale in setting the VS register. The "Units of Measurement in `nroff` and `troff`" table explains the units of measurement.

Table 8-4: Units of Measurement in `nroff` and `troff`

Unit	Abbr	`nroff`	`troff`
point	p	1/72 inch	1/72 inch
pica	p	1/6 inch	1/6 inch
em	m	width of one character	distance equal to number of points in the current typesize
en	n	width of one character	half an em
vertical space	v	amount of space in which each line of text is set, measured baseline to baseline	same
inch	i	inch	inch
centimeter	c	centimeter	centimeter
machine unit	u	1/240 inch	1/432 inch

The units ***point***, ***pica***, ***em***, and ***en*** are units of measurement used by tradition in typesetting. The ***vertical space*** unit also corresponds to the typesetting term ***leading***, which refers to the distance from the baseline of one line of type to the baseline of the next. Em and en are particulary interesting in that they are proportional to the type size currently in use (normally expressed as a number of points). An em is the distance equal to the number of points in the type size (roughly the width of the letter 'm' in that point size), while an en is half that (about the width of the letter 'n'). These units are convenient for specifying dimensions such as indentation. In `troff`, em and en have their traditional meanings, that is one em of distance is equal to two ens. For `nroff`, on the other hand, em and en both mean the same quantity of distance, the width of one typewritten character.

The ***machine unit*** is a special unit of dimension used by `nroff` and `troff` internally. This is the unit to which the programs convert almost all dimensions when storing them in memory, and is included here primarily for completeness. In using the features of —**ms**, it is sufficient to know that such a unit of measure exists.

Note that a change to a number register such as LL does not immediately change the related dimension at that point in the output. Instead, in the case of the line length for example, the change takes place at the beginning of the next paragraph, where —**ms** resets various dimensions to the current values of the related number registers.

If you need the effect immediately, use the normal ***troff*** command in addition to changing the number register. For example, to control the vertical spacing immediately, use:

```
.vs
```

This takes effect at the place where it occurs in your input file. Since it does not change the VS register, however, its effect lasts only until the beginning of the next paragraph. As a general rule, to make a permanent change, or one that will last for several paragraphs until you want to change it again, alter the value of the —**ms** register. If the change must happen immediately, somewhere other than the point shown in the table, use the `troff` request. If you want the change to be both immediate and lasting, do both.

Table 8-5: Summary of —**ms** Number Registers

Register	Controls	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. Use the `troff` `.ds` (define string) request to alter the string registers, as you use the `.nr` request for number registers. The page number on ***output*** is taken from register PN, to permit changing its output style. For more complicated headers and footers, you can redefine the macros PT and BT, as explained earlier. See the "Register Names" section for a full list.

8.6. Using `nroff` and `troff` Requests

You can use a small subset of the `troff` requests to supplement the —**ms** macro package.

Use `.nr` and `.ds` requests to manipulate the —**ms** number and string registers as described in the "Modifying Default Features" section. You can also freely use the other following requests in a file for processing with the —**ms** macro package. They all work with both typesetter and workstation or terminal output.

`.ad b` Adjust both margins. This is the default adjust mode.

`.bp` Begin new page.

`.br` 'Break' line; start a new output line whether or not the current one has been completely filled with text.

`.ce` ***n*** Center the following ***n*** input text lines individually in the output. If ***n*** is omitted, only the next (one) line of text is centered.

`.ds` *XX* Define string register named *XX*.

`.na` Turn off adjusting of right margins to produce ragged right.

`.nr` *XX* Define number register named XX.

`.sp` ***n*** Insert ***n*** blank lines. If ***n*** is omitted, one blank line is produced (the current value of the unit *v*). You can attach a unit of dimension to ***n*** to specify the quantity in units other than a number of blank lines.

Note: The macro package executes sequences of `troff` requests on its own, in a manner invisible to you. By inserting your own `troff` requests, you run the risk of introducing errors. The most likely result is simply for your `troff` requests to be ignored, but in some cases the results can include fatal `troff` errors and garbled typesetter output.

As a simple example, if you try to produce a centered heading with the input:

```
.ce
.SH
Text of section heading
```

you will discover that the heading comes out left-adjusted; the `.SH` macro, appearing after the `.ce` request overrules it and forces left-adjusting. But consider the following sequence:

```
.sp
.ce
.B
Line of text
```

which successfully produces a centered, boldface heading preceded by one line of vertical space. There are lots of tricks like this, so be careful.

To learn more about `troff` see the chapter on "Formatting Documents with `nroff` and `troff`".

8.7. Using —ms with `tbl` to Format Tables

Similar to the `eqn` macros are the macros `.TS` and `.TE` defined to separate tables from text with a little space (see the chapter "Formatting Tables with `tbl`"). 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.

8.8. Using —ms with `eqn` to Typeset Mathematics

If you have to print Greek letters or mathematical equations, see the chapter "Typesetting Mathematics with `eqn`" for equation setting. To aid *eqn* users, —**ms** provides definitions of `.EQ` and `.EN` which normally center the equation and set it off slightly. An argument to `.EQ` is taken to be an equation number and 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).

8.9. Register Names

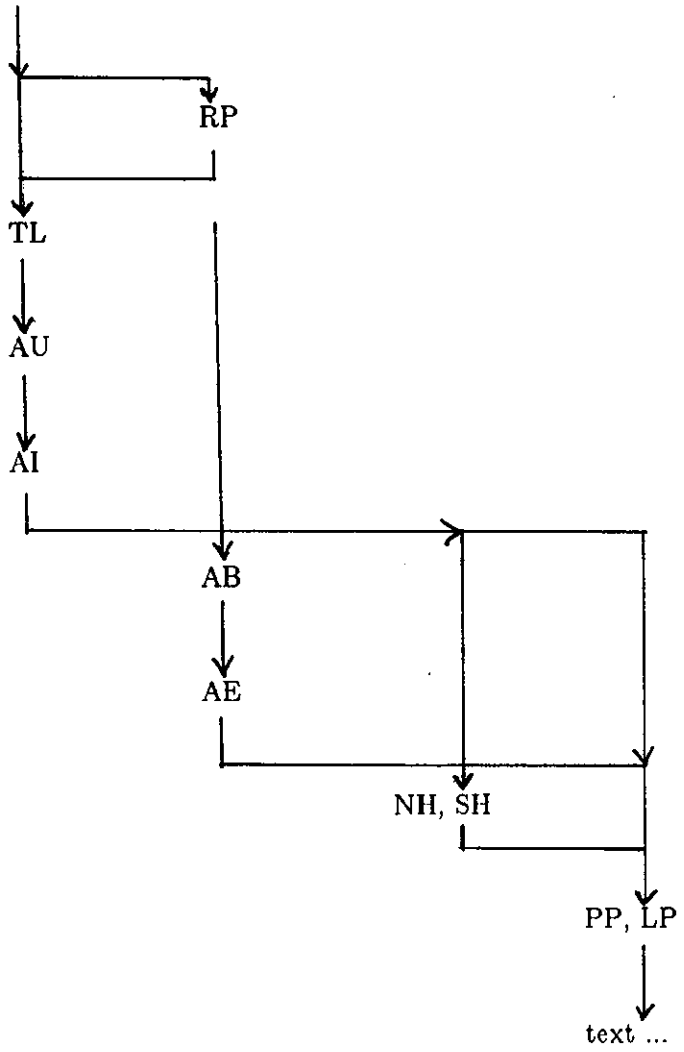
The —**ms** macro package uses the following register names internally. Independent use of these names in your own macros may produce incorrect output. Note that there are no lower-case letters in any —**ms** internal name.

Number Registers Used in —**ms**										
:	DW	GW	HM	IQ	LL	NA	OJ	PO	T.	TV
#T	EF	H1	HT	IR	LT	NC	PD	PQ	TB	VS
T.	FC	H2	IF	IT	MF	ND	PE	PS	TC	WF
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 —**ms**										
'	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

8.10. Order of Requests in Input

The following diagram provides a quick reference on how to order requests when using the **—ms** macro package to format a document in released format. For simpler documents, start with an `.LP` initializing request.



8.11. —ms Request Summary

This section includes tables of the old Bell Laboratories that have been removed from the new —**ms** package, of new —**ms** requests and string definitions, and of useful printing and displaying commands. It also includes a complete —**ms** request and string summary for easy reference.

Table 8-6: Bell Laboratories Macros (deleted from —**ms**)

Macro Request	Explanation
.CS	Cover sheet
.EG	BTL Engineer's Notes
.HO	Bell Labs, Holmdel, N.J.
.IH	Bell Labs, Naperville, Ill.
.IM	BTL internal memo
.MF	BTL file memo
.MH	Bell Labs, Murray Hill, N.J.
.MR	BTL record memo
.ND	BTL date
.OK	BTL keywords for tech memo
.PY	Bell Labs, Piscataway, N.J.
.SG	Signatures for tech memo
.TM	BTL technical memo
.TR	BTL report format
.WH	Bell Labs, Whippany, N.J.

Table 8-7: New —**ms** Requests

Macro Request	Explanation
.AM	New accent mark definitions.
.CT	Chapter title in . TM format.
.EH	Define even three-part page header.
.EF	Define even three-part page footer.
.FE	End automatically numbered footnote.
.FS	Begin automatically numbered footnote.
.IP **	Number endnotes.
.IX	Index words.
.OF	Define odd three-part page footer.
.OH	Define odd three-part page header.
.P1	Put header on page one in . TM format.
.PX	Print table of contents.
.TM	Thesis format.
.XS	Start table of contents entry.
.XE	End table of contents entry.
.XA	Additional table of contents entry.
.PX	Prints table of contents.
.XP	Exdented paragraph.

Table 8-8: New String Definitions

Definition	In nroff	In troff
*-	Two hyphens --	Em dash —
*Q	Open quote ``	Open quote “
*U	Closed quote ''	Closed quote ”

Table 8-9: —**ms** Macro Request Summary

Macro Request	Initial Value	Cause Break?	Explanation
.1C	yes	yes	One column format on a new page.
.2C	no	yes	Two column format.
.AB	no	yes	Begin abstract.
.AE	—	yes	End abstract.
.AI	no	yes	Author's institution follows.
.AM	—	no	New accent mark definitions
.AT	no	yes	Print '...Attached' and turn off line filling.
.AU	no	yes	Author's name follows.
.B *x*	no	no	Print *x* in boldface; if no argument switch to boldface.
.B1	no	yes	Begin text to be enclosed in a box.
.B2	no	yes	End text to be boxed and print it.
.BT	date	no	Bottom title, automatically invoked at foot of page. May be redefined.
.BX *x*	no	no	Print *x* in a box.
.CM	-	no	Cut mark between pages (only if troff).
.CT	-	yes	Chapter title in thesis mode only. Page number moved to CF.
.DA *x*	date	no	'Date line' at bottom of page is *x* (only in nroff). Default is today.
.DE	—	yes	End displayed text. Implies .KE.
.DS *x*	no	yes	Start of displayed text to appear verbatim liine–by–line. *x*=I for indented display (default), *x*=L for left-adjusted on the page, *x*=C for centered, *s*=B for make left–justified block, then center whole block. Implies .KS.
.EF *x*	—	no	Even three-part page footer *x*
.EN	—	yes	Space after equation produced by ***eqn*** or ***neqn***.

Macro Request	Initial Value	Cause Break?	Explanation
.EQ *x y*	—	yes	Precede equation; break out and add space. Equation number is *y*. The optional argument *x* may be I to indent equation (default), L to left–adjust the equation, or C to center it.
.FE	—	yes	End footnote.
.FS *x*	—	no	Start footnote. *x* is optional footnote label. The note will be printed at the bottom of the page.
.I *x*	no	no	Italicize *x*; if *x* is missing, italic text follows.
.IP *x y*	no	yes	Start indented paragraph, with hanging tag *x*. Indentation is *y* ens (default 5).
.KE	—	yes	End keep. Put kept text on next page if not enough room.
.KF	no	yes	Start floating keep. If the kept text must be moved to the next page, float later text back to this page.
.KS	no	yes	Start keeping following text.
.LG	no	yes	Make letters larger.
.LP	yes	yes	Start left-blocked paragraph.
.ND *date*	—	no	Use date supplied if any as page footer; only in special format positions.
.NH *n*	—	yes	Same as .SH with section number supplied automatically. Numbers are multilevel, like 1.2.3, where *n* tells what level is wanted (default is 1).
.NL	yes	no	Make letters normal size.
.IX *x y*	—	yes	Index entries *w* and *y* and so on up to 5 levels. Make letters normal size.
.OF *x*	—	no	Odd three-part page footer.
.OH *header*	—	no	Odd three-part page header.
.P1	—	no	Print header on first page (only in thesis mode).
.PP	no	yes	Begin paragraph. First line indented.
.PT	pg #	—	Page title, automatically invoked at top of page. May be redefined.
.PX *x*	—	yes	Print table of contents; *x*=no suppresses title.
.QP	—	yes	Begin single paragraph which is indented and shorter.
.R	yes	no	Roman text follows.

Macro Request	Initial Value	Cause Break?	Explanation
.RE	–	yes	End relative indent level.
.RP	no	–	Cover sheet and first page for released paper. Must precede other requests.
.RS	–	yes	Start level of relative indentation. Following `.IP`s are measured from current indentation.
.SH	–	yes	Section head follows, font automatically bold.
.SM	no	no	Make letters smaller.
.TA *x*...	5...	no	Set tabs in ens. Default is 5 10 15 ...
.TE	–	yes	End table.
.TH	–	yes	End heading section of table.
.TL	no	yes	Title follows.
.TM	off	no	Thesis mode format.
.TS *x*	–	yes	Begin table; if *x* is H, table has repeated heading on subsequent pages.
.UL *x*	–	yes	Underline argument, even in `troff`.
.XA *x y*	–	yes	Another index entry; *x*=page for no for none, *y*=indent.
.XE	–	yes	End index entry or series of `.IX` entries.
.XS *x y*	–	yes	Begin index entry; *x*=page or no for none, *y*=indent.
.UL *x*	–	yes	Underline argument, even in `troff`.

Table 8-10: —**ms** String Definitions

Name	Definition	In nroff	In troff
quote	*Q	”	“
unquote	*U	”	”
dash	*-	—	—
month of year	*(MO	April	April
current date	*(DY	April 6, 1985	April 6, 1985
automatically-numbered footnote	**		

The following table summarizes command lines you use to print and display documents. Use the same order with `troff` for preprocessing files with `tbl` and `eqn`.

If you use the two-column `.2C` request, either pipe the `nroff` output through `col` or make the first line of the input `.pi /usr/bin/col`.

Table 8-11: Printing and Displaying Documents

What You Want to Do	How to Do it
Display a file	**nroff** **-ms** *file* ... **|** **more**
Print a file on the line printer	**nroff** **-ms** *file* **|** **lpr** *-printer*
Print a file with tables	**tbl** *file* **|** **nroff** **-ms** **|** **lpr** *-printer*
Print a file with equations	**neqn** *file* **|** **nroff** **-ms** **|** **lpr** *-printer*
Print a file with tables and equations	**tbl** *file* **|** **neqn** **|** **nroff** **-ms** **|** **lpr** *-printer*
Print a file document with `troff`	**troff** **-ms** *file* **|** **lpr** **-t** *-printer*

Chapter 9

The –man Macro Package

The **–man** macro package is used to format the manual pages to look like those in the *Commands Reference Manual for the Sun Workstation*, for example.

9.1. Parts of a Manual Page

A manual page consists of several parts:

- The first part is the `.TH` line. This line identifies the manual page and sets up the titles and other information to print the page headers and footers.
- The next few sections are all introduced by `.SH` macro requests.

A skeleton command file would look something like this:

```
.TH XX 1 "7 November 1984"
.SH NAME
.SH SYNOPSIS
.SH DESCRIPTION
.SH OPTIONS
.SH FILES
.SH "SEE ALSO"
.SH DIAGNOSTICS
.SH BUGS
```

The sections have the following meanings:

NAME	The name of the command and a short description.
SYNOPSIS	A short synopsis of the command and its options and arguments.
DESCRIPTION	A brief narrative description of what the command does.
OPTIONS	A list of the options in terse itemized list format.
FILES	Names of files that this command uses or creates.
SEE ALSO	Other relevant commands and files and so on
BUGS	Known deficiencies in the command.

Occasionally there may be other sections you can add. For instance, a couple of the manual pages have a section called RESTRICTIONS, which contains the notice that this software is not distributed outside of the United States of America.

Leave out sections that do not apply — it is not necessary to have a title without any content to go with it. Definitely avoid sections that read:

BUGS
None.

9.2. Coding Conventions

The following subsections compose a fairly detailed description of what the different sections of the manual page contain.

9.2.1. The .TH *Line — Identifying the Page*

The **.TH** macro is the macro that identifies the page. The format is

.TH *n c x v m*

This means, for example: Begin page named *n* of chapter *c*. The *x* argument is for extra commentary for the center page footer. The *v* argument alters the left portion of the page footer. The *m* argument alters the center portion of the page header. The .TH command line also incidentally sets the prevailing indent and tabs to .5i.

To code a manual page called ***troff***(1), for example, you would code a .TH macro like:

.TH TROFF 1 "*today's date*"

The third parameter to the **.TH** macro is the date on which you created or last changed the manual page. You code *today's date* in the form

numerical day spelled-out month numerical year

So if today is September 3rd, 1984, you code the **.TH** macro like:

```
.TH TROFF 1 "3 September 1984"
```

This form of coding the date ensures that people who do not live in the United States are not confused by a date written in the form 9/3/84 and think that this means the 9th of March instead of the 3rd of September.

9.2.2. The NAME *Line*

The **NAME** line is a one-liner that identifies the command or program. You code the information like this:

```
.SH NAME
troff \- typeset or format documents
```

This line must be typed all in the Roman font with no font changes or point-size changes or any other text manipulation. Typing the command line all in Roman with no text manipulation is for the permuted index generator. It gets all confused if there is anything in that line other than plain text.

Note the \- in there — why do we type a \-? Well, in ***troff*** jargon, a simple - sign gets you a hyphen. We actually would like a en-dash (like −) instead of a hyphen, in lieu of actually having a em-dash (like —). This use of the \- to get a − is a UNIX† tradition.

9.2.3. The **SYNOPSIS** *Section*

The **SYNOPSIS** line(s) show the user what options and arguments can be typed. The conventions for the SYNOPSIS have varied wildly over the years. Nonetheless, here are the guidelines:

- ***Literal text*** (that is, what the user types) is coded in **bold face**.
- ***Variables*** (that is, things someone might ***substitute for***) are typed in ***italic text***.
- ***Optional things*** are enclosed in brackets — that is the characters [and].
- ***Alternatives*** are separated by the vertical bar sign (|).

The synopsis should show what the options are — some manual pages used to read like this:

SYNOPSIS
troff [***options***] filename ...

but it ***should*** read:

SYNOPSIS
troff [**−o***pagelist*] [**−n***N*] [**−m** ***name***] ... [filename]

9.2.4. The **DESCRIPTION** *Section*

The **DESCRIPTION** section of a manual page should contain a brief description of what the command does ***for the user***, in terms that the user cares about.

Within the DESCRIPTION and OPTIONS sections, ***italic text*** is used for filenames and command names. The rationale here is that UNIX commands are simply files. When referring to other manual pages, you type the name in italics and the following parenthesized section number in Roman, as in ***make***(1). Use the **−man** macro .IR to get alternating words joined in italic and Roman fonts. Note that the macros that join alternating words in different fonts (.IR, .IB, .BR, .BI, .RI, .RB) all accept only ***six*** parameters. See the section on how to format a manual page for more formatting rules.

Part of the description in the ***grep*** manual page used to read:

---

[4] † UNIX is a trademark of Bell Laboratories.

..... ***grep*** patterns are limited regular expressions in the style of ***ed***(1); it uses a compact nondeterministic algorithm. ***egrep*** patterns are full regular expressions; it uses a fast deterministic algorithm that sometimes needs exponential space. ***fgrep*** patterns are fixed strings; it is fast and compact.

Most users do not care that egrep uses a fast deterministic algorithm. As an example of a more useful way of describing a command for the user, here is how that sentence in the ***grep*** manual page currently reads.

..... ***grep*** patterns are limited regular expressions in the style of ***ed***(1). ***egrep*** patterns are full regular expressions including alternation. ***fgrep*** searches for lines that contain one of the (newline-separated) ***strings***. ***fgrep*** patterns are fixed strings — no regular expression metacharacters are supported.

Here's another bad example: the ***lpr***(1) command used to tell you that the **−s** option uses the ***symlink***(2) system call to make a symbolic link to the data file instead of copying the data file to the spool area. The user may not know what this means or how to use the information. The description was changed to just tell you that the **−s** option makes a symbolic link to the data file. How it is done is of little concern to some poor blighter who just wants to print a file.

9.2.5. The **OPTIONS** *Section*

The **OPTIONS** section of a manual page contains an itemized list of the options that the command recognizes, and how the options affect the behavior of the command. The general format for this section is

−option Description of what the option does.

A specific example from the ***troff*** manual page looks like this:

OPTIONS

Options may appear in any order as long as they appear **before** the files.

−o*list*
Print only pages whose page numbers appear in the comma-separated list of numbers and ranges. A range N-M means pages N through M; an initial -N means from the beginning to page N; and a final N- means from N to the end.

−n*N*
Number first generated page *N*.

−m*name*
Prepend the macro file /usr/lib/tmac/tmac.name to the input files.

−r*aN*
Set register ***a*** (one-character) to *N*.

−i
Read standard input after the input files are exhausted.

−q
Invoke the simultaneous input-output mode of the ***rd*** request.

−t
Direct output to the standard output instead of the phototypesetter. In general, you will have to use this option if you don't have a typesetter attached to the system.

−a
Send a printable ASCII approximation of the results to the standard output.

Some options of ***troff*** only apply if you have a C/A/T typesetter attached to your system. These options are here for historical reasons:

−s*N*
Stop every *N* pages. ***troff*** stops the phototypesetter every *N* pages, produces a trailer to allow changing cassettes, and resumes when the typesetter's start button is pressed.

−f
Refrain from feeding out paper and stopping phototypesetter at the end of the run.

−w
Wait until phototypesetter is available, if currently busy.

-b
Report whether the phototypesetter is busy or available. No text processing is done.

−p*N*
Print all characters in point size *N* while retaining all prescribed spacings and motions, to reduce phototypesetter elasped time.

9.2.6. The **FILES** *Section*

The **FILES** section of a manual page contains a list of the files that the program accesses, creates, or modifies. Obviously, you can leave this section out if the program uses no files.

The example from the ***troff*** manual page looks like this:

> If the file /usr/adm/tracct is writable, ***troff*** keeps phototypesetter accounting records there. The integrity of that file may be secured by making ***troff*** a 'set user-id' program.

FILES

/tmp/ta*	temporary file
/usr/lib/tmac/tmac.*	standard macro files
/usr/lib/term/*	terminal driving tables for nroff
/usr/lib/font/*	font width tables for troff
/dev/cat	phototypesetter
/usr/adm/tracct	accounting statistics for /dev/cat

9.2.7. The **SEE ALSO** *Section*

The **SEE ALSO** section of a manual page contains a list of references to other programs, files, and manuals relating to this program. For example, on the ***troff*** manual page, the **SEE ALSO** section looks like this:

SEE ALSO
Formatting Documents with nroff and troff in Editing and Text Processing on the Sun Workstation
nroff(1), eqn(1), tbl(1), ms(7), me(7), man(7), col(1)

Contrary to the convention used everywhere else, the names in this section do not follow the convention of italic name followed by Roman section number — you just type the whole thing like make(1) all in Roman. This is a UNIX ***tradition***.

Make sure that the references are ***useful*** — the ***rm***(1) command references the ***unlink***(2) system call. Does the user ***care*** what system call is used to get rid of a file? It's not intuitive that you use a function called ***unlink*** to remove a file.

Leave this section out if there are no interesting references.

9.2.8. The **BUGS** *Section*

The **BUGS** section of a manual page is supposed to convey limitations of the command or bad behavior of the command to the reader. Please limit bugs to these categories. Too often, the **BUGS** section contains WIBNIs. A WIBNI is a 'Wouldn't It Be Nice If...' — this is not the place for them.

Leave this section out altogether if there are no bugs worth noting.

9.3. New Features of the –man Macro Package

9.3.1. New Number Registers

Recent enhancements to the **–man** macro package facilitate including manual pages in manuals. The major new features are number registers that can be set from the ***itroff***, ***iroff***, ***troff***, ***ditroff***, or ***nroff*** command line. The number registers are:

D	Format the document for double-sided printing if the **D** number register is set to 1. Double-sided printing means that the page numbers appear in different locations on odd and even pages. Page numbers appear in the running footers in the lower ***right*** corner of ***odd-numbered*** pages and in the lower ***left*** corner of ***even-numbered*** pages.
C	Number pages contiguously — pages are numbered 1, 2, 3,... even when you format more than one manual page at a time. Every new topic used to start numbering at page 1.
P*nnn*	Start Page numbering at page ***nnn*** — page numbering starts at page 1 if not otherwise specified.

X*nnn* Number pages as ***nnn*a**, ***nnn*b**, etc when the current page number becomes ***nnn***. This feature is for generating update pages to slot in between existing pages. For example, if a new page called ***skyversion***(8) should be included in an interim release, we can number that page as page '26a' and drop it into the existing manual in a reasonable fashion.

9.3.2. Using the Number Registers

Number registers are set from the ***itroff***, ***iroff***, ***troff***, ***ditroff***, or ***nroff*** command line by the **−r** (set register) option, followed immediately by the ***one-letter*** name of the register, followed immediately by the value to put into the number register:

```
hostname% /usr/local/itroff -man -rD1 manpage.1
hostname%
```

This example shows how to request a format suitable for double-sided printing.

If your ***grab***(1) manual page used to be three pages long and is now five pages long, you need the pages numbered 1, 2, 3, 3a, and 3b instead of 1, 2, 3, 4, and 5. You get this effect by using the **−rX** option on the command line, setting the **X** register to 3:

```
hostname% /usr/local/itroff -man -rX3 grab.1
hostname%
```

We introduced the ***screendump***(1) and ***screenload***(1) manual pages in the 1.2 release. ***screendump***(1) and ***screenload***(1) come immediately after the ***sccsdiff***(1) manual page. ***sccsdiff***'s last page number is page 260, so we get ***screendump***(1) and ***screenload***(1) formatted with this command to start page numbering at 260 and to start putting in extra page letters at 260 as well:

```
hostname% /usr/local/itroff -man −rP260 -rX260 screendump.1 screenload.1
hostname%
```

9.4. How to Format a Manual Page

Any text argument *t* to a macro request may be from zero to six words. Quotes my be used to include blanks in a 'word'. If the text field is empty, the macro request is applied to the next input line with text to be printed. In this way, **.I** italicizes an entire line, and **.SM** followed on a separate line by **.B** creates small, bold letters.

A prevailing indent distance is remembered between successive indented paragraphs, and is reset to the default value upon reaching a non-indented paragraph. Default units for indents *i* are ens.

Type font and size are reset to the default values before each paragraph, and after processing font- and size-setting macros.

These strings are predefined by **–man**:

*R ®, '(Reg)' in *nroff*.
*S Change to default type size.

9.5. Summary of the –man Macro Package Requests

Request	Cause Break	If no Argument	Explanation
.B *t*	no	*t*=next text line	Text *t* is bold.
.BI *t*	no	*t*=next text line	Join words of *t* alternating bold and italic.
.BR *t*	no	*t*=next text line	Join wordsof *t* alternating bold and Roman.
.DT	no	.5i 1i...	Restore default tabs.
.HP *i*	yes	*i*=prevailing indent	Set prevailing indent to *i*. Begin paragraph with hanging indent.
.I *t*	no	*t*=next text line	Text *t* is italic.
.IB *t*	no	*t*=next text line	Join words of *t*alternating italic and bold.
.IP *x i*	yes	*x*=""	Same as .TP with tag *x*.
.IR *t*	no	*t*=next text line	Join words of *t* alternating italic and Roman.
.LP	yes		Same as .PP.
.PD *d*	no	*d*=.4v	Interparagraph distance is *d*.
.PP	yes		Begin paragraph. Set prevailing indent to .5i.
.RE	yes		End of relative indent. Set prevailing indent to amount of starting .RS.
.RB *t*	no	*t*=next text line	Join words of *t* alternating Roman and bold.
.RI *t*	no	*t*=next text line	Join words of *t* alternating Roman and italic.
.RS *i*	yes	*i*=prevailing indent	Start relative indent, move left margin in distance *i*. Set prevailing indent to .5i for nested indents.
.SH *t*	yes	*t*=next text line	Subheading.
.SM *t*	yes	*t*=next text line	Text *t* is two point sizes smaller than surrounding text.

Request	**Cause** Break	**If no** Argument	**Explanation**
.TH *n c x v m*	yes		Begin page named *n* of chapter *c*. The *x* argument is for extra commentary for the center page footer. The *v* argument alters the left portion of the page footer. The *m* argument alters the center portion of the page header. The .TH command line also incidentally sets the prevailing indent and tabs to
.TP *i*	yes	*i*=prevailing indent	Set the prevailing indent to *i*. Begin indented paragraph with hanging tag given by the next text line. If the tag does not fit, place it on a separate line.

To learn how to format manual pages on your terminal or workstation screen, refer to the

Chapter 10

Formatting Tables with tbl

This chapter[5] provides instructions for preparing ***tbl*** input to format tables and for running the ***tbl*** preprocessor on a file. It also supplies numerous examples after which to pattern your own tables. The description of instructions is precise but technical, and the newcomer may prefer to glance over the examples first, as they show some common table arrangements.

Tbl turns a simple description of a table into a ***troff*** or ***nroff*** program that prints the table. From now on, unless noted specifically, we'll refer to both ***troff*** and ***nroff*** as ***troff*** since ***tbl*** treats them the same. ***Tbl*** makes phototypesetting tabular material relatively simple compared to normal typesetting methods. You may use ***tbl*** with the equation formatting program ***eqn*** or various layout macro packages, as ***tbl*** does not duplicate their functions.

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

The input to ***tbl*** is text for a document, with the text 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

[5] **The material in this chapter is derived from *Tbl — A Program to Format Tables*, M.E. Lesk, Bell Laboratories, Murray Hill, New Jersey.**

copied, too, so that ***troff*** page layout macros, such as the formatting macros, can use these lines to delimit and place tables as necessary. In particular, any arguments on the `.TS` or `.TE` lines are copied but otherwise ignored, and may be used by document layout macro commands.

The format of the input is as follows:

```
. . .
ordinary text of your document
. . .
.TS
first table
.TE
. . .
ordinary text of your document
. . .
.TS
second table
.TE
. . .
ordinary text of your document
. . .
```

where the format of each table is as follows:

```
.TS
options for the table ;
format describing the layout of the table .
data to be laid out in the table
                    .
                    .
                    .
data to be laid out in the table
.TE
```

Each table is independent, and must contain formatting information, indicated by ***format describing the layout of the table,*** followed by the ***data to be laid out in the table.*** You may precede the formatting information, which describes the individual columns and rows of the table, by ***options for the table*** that affect the entire table.

10.1. Running tbl

You can run *tbl* on a simple table by piping the *tbl* output to ***troff*** (or your installation's equivalent for the phototypesetter) with the command:

```
tutorial% tbl file | troff -options
```

where ***file*** is the name of the file you want to format. For more complicated use, where there are several input files, and they contain equations and ***−ms*** macro package requests as well as tables, the normal command is:

```
tutorial% tbl file1 file2 . . . | eqn | troff -ms
```

You can, of course, use the usual options on the ***troff*** and ***eqn*** commands. The usage for ***nroff*** is similar to that for ***troff***, but only printers such as the TELETYPE® Model 37 and Diablo-mechanism (DASI or GSI) or other printers that can handle reverse paper motions can print boxed tables directly. If you are running ***tbl*** on a line printer that does not filter reverse paper motions, use the ***col*** processor to filter the multicolumn output.

If you are using an IBM 1403 line printer 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, ***tbl*** accepts them as well.

Caveats: Note that when you use ***eqn*** and ***tbl*** together on the same file, put ***tbl*** 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 there are equations within tables, using the ***delim*** mechanism in ***eqn***, you must put ***tbl*** first or the output will be scrambled. Also, beware of using equations in **n**-style columns; this is nearly always wrong, since ***tbl*** attempts to split numerical format items into two parts, and this is not possible with equations. To avoid this, use the **delim(*xx*)** table option to prevent splitting 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. Avoid using ***troff*** number registers used by ***tbl*** 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 that you may not use them 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. Use of this macro in the page footer boxes a multi-page table. In particular, you can use the **−ms** macros 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 there aren't any. 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. For example:

```
.TS H
center box tab (/);
c s
l l .
Employees
_
Name/Phone
_
.TH
Jonathan Doe/123-4567
< etc. >
.TE
```

Note that this is ***not*** a feature of ***tbl***, but of the **−ms** layout macros.

10.2. Input Commands

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 sections that follow explain how to enter the various parts of the table.

10.2.1. Options that Affect the Whole Table

There may be a single line of options affecting the whole table. If present, this line must follow the '.TS' line immediately, 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-adjusted).
`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 — a frame.
`tab(`*x*`)`	use *x* instead of tab to separate data items.
`linesize(`*n*`)`	set lines or rules (such as from `box`) in *n* point type.
`delim(`*xy*`)`	recognize *x* and *y* as the *eqn* delimiters.

A standard option line is:

```
center box tab (/) ;
```

which centers the table on the page, draws a box around it, and uses the slash '/' character as the column separator for data items.

The *tbl* program tries to keep boxed tables on one page by issuing appropriate *troff* 'need' (`.ne`) commands. These requests are calculated from the number of lines in the tables, so if there are spacing commands embedded in the input, these requests may be inaccurate. Use normal *troff*

procedures, such as keep-release macros, in this case. If you must have a multi-page boxed table, use macros designed for the purpose, as explained above under ***Running 'tbl'.***

10.2.2. Key Letters — Format Describing Data Items

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 present as shown below. 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, tabs, or a visible character such as a slash '/'. Each key-letter is one of the following: "1 — left adjusted"

`L` or `l` indicates a left-adjusted column entry.

`R` or `r` indicates a right-adjusted column entry.

`C` or `c` indicates a centered column entry.

`N` or `n` indicates a numerical column entry, to line up the units digits of numerical entries.

`A` or `a` indicates an alphabetic subcolumn; all corresponding entries are aligned on the left, and positioned so that the widest is centered within the column (see the 'Some London Transport Statistics' example).

`S` or `s` indicates a spanned heading; that is, it indicates that the entry from the previous column continues across this column; not allowed for the first column.

`^` indicates a vertically spanned heading; that is, it indicates that the entry from the previous row continues down through this row; not allowed for the first row of the table.

When you specify numerical alignment, ***tbl*** requires a location for the decimal point. 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, you may use the special non-printing character string `\&` to override unconditionally 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 (we use `L` here instead of `l` for readability; they have the same meaning as key-letters). Alignment within the numerical items is preserved. This is similar to the way `a` type data are formatted, as explained above. However, alphabetic subcolumns (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.

Note: Do not use the **n** and **a** items in the same column.

For readability, separate the key-letters describing each column with spaces. Indicate the end of the format section 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 is:

```
.TS
c  s  s
l  n  n .
text
.TE
```

which specifies a table of three columns. The first line of the table contains a centered heading that spans 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 is:

	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

10.2.3. Optional Features of Key Letters

There may be extra information following a key-letter that modifies its basic behavior. Additional features of the key-letter system follow:

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. You can also type this in the data portion. 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 displayed.

Vertical lines
— A vertical bar may be placed between column key-letters. This draws 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.

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*** (one en is about the width of the letter 'n')[6]. If the 'expand' option is used, 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, that is the largest space requested, governs.

[6] **More precisely, an en is a number of points (1 point = 1/72 inch) equal to half the current type size.**

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` or `T`, any corresponding vertically spanned item begins at the top line of its range.

Font changes
— A key-letter may be followed by a string containing a font name or number preceded by the letter `f` or `F`. This indicates that the corresponding column should be in a different font from the default font, which is usually Roman. 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.

Point size changes
— A key-letter may be followed by the letter `p` or `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.

Vertical spacing changes
— A key-letter may be followed by the letter `v` or `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 ***Text Blocks*** below).

Column width indication
— A key-letter may be followed by the letter `w` or `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 considered 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; if none is used, the default is ens. If the width specification is a unitless integer, you may omit the parentheses. If the width value is changed in a column, the ***last*** one given controls.

Equal width columns
— A key-letter may be followed by the letter `e` or `E` to indicate equal width columns. All columns whose key-letters are followed by `e` or `E` are made the same width. In this way, you can format a group of regularly spaced columns.

Note:
The 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)fI  6
```

Alternative notation
— Instead of listing the format of successive lines of a table on consecutive lines of the format section, separate successive line formats on the same line by commas. The format for the sample table above can be written:

```
c s s, l n n .
```

Default

— 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.

10.2.4. Data to be Formatted in the Table

Type the data for the table after the format line. Normally, each table line is typed as one line of data. Break very long input lines by typing a backslash ' \' as a continuation marker at the end of the run-on line. That line is combined with the following line upon formatting and the ' \' vanishes. The data for different columns, that is, the table entries, are separated by tabs, or by whatever character has been specified in the option ***tabs*** option. We recommend using a visible character such as the slash character '/'. There are a few special cases:

Troff commands within tables

— An input line beginning with a '.' followed by anything but a number is assumed to be a command to ***troff*** and is passed through unchanged, retaining its position in the table. So, for example, you can produce space within a table by '.sp' commands in the data.

Full width horizontal lines

— An input ***line*** containing only the character '_' (underscore) or '=' (equal sign) represents a single or double line, respectively, extending the full width of the ***table***.

Single column horizontal lines

— An input table ***entry*** containing only the character '_' or '=' represents a single or double line extending the full width of the ***column***. Such lines are extended to meet horizontal or vertical lines adjoining this column. To obtain these characters explicitly in a column, either precede them by '\&' or follow them by a space before the usual tab or newline.

Short horizontal lines

— An input table ***entry*** containing only the string '_' represents a single line as wide as the contents of the column. It is not extended to meet adjoining lines.

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 '^'.

Text blocks

— To include a block of text as a table entry, precede it by T{ and follow it by T}. To enter, as a single entry in the table, something that cannot conveniently be typed as a simple string between tabs, use:

```
. . . T{
block of text
T} . . .
```

Note that the T} end delimiter must begin a line; additional columns of data may follow after a tab on the same line. See the 'New York Area Rocks' example for an illustration of included text blocks in a table. If you use more than twenty or thirty text blocks in a table, various limits in the ***troff*** program are likely to be exceeded, producing diagnostics such as

'too many text block diversions.'

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 $L \times C/(N+1)$ 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 you can put any number of lines in a table, only the first 200 lines are used in calculating the widths of the various columns. Arrange a multi-page table 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\fIdata\fP\s0`). Therefore, although arbitrary ***troff*** requests may be sprinkled in a table, use requests such as `.ps` (set the point size) with care to avoid confusing the width calculations.

10.2.5. Changing the Format of a Table

If you must change the format of a table after many similar lines, as with sub-headings or summarizations, use the `.T&` (table continue) command to change column parameters. The outline of such a table input is:

```
.TS                                          start of the table
options afecting the whole table;
format of the columns .
data to be formatted in the table
.
.
.
data to be formatted in the table
.T&                                          indicates a new format for the table
format of the columns .
data to be formatted in the table
.
.
.
data to be formatted in the table
.T&                                          indicates a new format for the table
format of the columns .
data to be formatted in the table
.
.
.
data to be formatted in the table
.TE                                          end of the table
```

as in the 'Composition of Foods' and 'Some London Transport Statistics' examples. 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.

10.3. Examples

Here are some examples illustrating features of *tbl*. Glance through them to find one that you can adapt to your needs.

Although you can use a tab to separate columns of data, a visible character is easier to read. The standard column separator here is the slash '/'. If a slash is part of the data, we indicate a different separator, as in the first example.

Input:

```
.TS
 tab (%) box;
c c c
l l l.
Language%Authors%Runs on

Fortran%Many%Almost anything
PL/1%IBM%360/370
C%BTL%11/45,H6000,370
BLISS%Carnegie-Mellon%PDP-10,11
IDS%Honeywell%H6000
Pascal%Stanford%370
.TE
```

Output:

Language	Authors	Runs on
Fortran	Many	Almost anything
PL/1	IBM	360/370
C	BTL	11/45,H6000,370
BLISS	Carnegie-Mellon	PDP-10,11
IDS	Honeywell	H6000
Pascal	Stanford	370

Input:

```
.TS
tab (/) allbox;
c s s
c c c
n n n.
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*
.TE
* (first quarter only)
```

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
tab (/) 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
```

Output:

Major New York Bridges		
Bridge	Designer	Length
Brooklyn Manhattan Williamsburg	J. A. Roebling G. Lindenthal L. L. Buck	1595 1470 1600
Queensborough	Palmer & Hornbostel	1182
Triborough	O. H. Ammann	1380 383
Bronx Whitestone Throgs Neck	O. H. Ammann O. H. Ammann	2300 1800
George Washington	O. H. Ammann	3500

Input:

```
.TS
tab (/) ;
c c
np-2 | n | .
/Stack
/_
1/46
/_
2/23
/_
3/15
/_
4/6.5
/_
5/2.1
/_
.TE
```

Output:

	Stack
1	46
2	23
3	15
4	6.5
5	2.1

Input:

```
.TS
tab (/) 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
```

Output:

january	february	march
april	may	
june	july	**Months**
august	september	
october	november	december

Input:

```
.TS
 tab (/) 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:

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
tab (/) allbox;
cfI 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:

New York Area Rocks

Era	Formation	Age (years)
Precambrian	Reading Prong	>1 billion
Paleozoic	Manhattan Prong	400 million
Mesozoic	Newark Basin, incl. Stockton, Lockatong, and Brunswick formations; also Watchungs and Palisades.	200 million
Cenozoic	Coastal Plain	On Long Island 30,000 years; Cretaceous sediments redeposited by recent glaciation.

Input:

```
.EQ
delim $$
.EN

. . .

.TS
tab (/) doublebox;
c c
l l.
Name/Definition
.sp
.vs +2p
Gamma/$GAMMA (z) = int sub 0 sup inf  t sup {z-1} e sup -t dt$
Sine/$sin (x) = 1 over 2i ( e sup ix - e sup -ix )$
Error/$ roman erf (z) = 2 over sqrt pi int sub 0 sup z e sup {-t sup 2} dt$
Bessel/$ J sub 0 (z) = 1 over pi int sub 0 sup pi cos ( z sin theta ) d theta $
Zeta/$ zeta (s) = sum from k=1 to inf k sup -s ~~( Re~s > 1)$
.vs -2p
.TE
```

Output:

Name	Definition
Gamma	$\Gamma(z)=\int_0^\infty t^{z-1}e^{-t}\,dt$
Sine	$\sin(x)=\frac{1}{2i}(e^{ix}-e^{-ix})$
Error	$\operatorname{erf}(z)=\frac{2}{\sqrt{\pi}}\int_0^z e^{-t^2}dt$
Bessel	$J_0(z)=\frac{1}{\pi}\int_0^\pi \cos(z\sin\theta)\,d\theta$
Zeta	$\zeta(s)=\sum_{k=1}^{\infty} k^{-s}$ (Re $s>1$)

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 & Leading for 10-Pt. 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 & Leading for 10-Pt. 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
tab (/) ;
c s
c|p-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./8,553
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
.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.	5,553
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

Input:

```
.ps 8
.vs 10p
.TS
tab (/) center box;
c s s
ci s s
c c c
1B 1 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
1B 1 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
(Democrats)

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
	(Republicans)	
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. Examine the appearance of centered tables or expanded tables, and observe how such tables are formatted.

Input:

```
.TS
center tab (/) ;
c s s s
c s s s
c c c c
n n n n.
LYKE WAKE WALK
Successful Crossings 1959–1966
Year/First Crossings/Repeats/Total
1959/89/23/112
1960/222/33/255
1961/650/150/800
1962/1100/267/1367
1963/1054/409/1463
1964/1413/592/2005
1965/2042/771/2813
1966/2537/723/3260
.TE
```

Output:

LYKE WAKE WALK
Successful Crossings 1959–1966

Year	First Crossings	Repeats	Total
1959	89	23	112
1960	222	33	255
1961	650	150	800
1962	1100	267	1367
1963	1054	409	1463
1964	1413	592	2005
1965	2042	771	2813
1966	2537	723	3260

Input:

```
.TS
tab (/) 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}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	
American Museum of Natural History	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
Bronx Zoo	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
Brooklyn Museum	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
New-York Historical Society	All the original paintings for Audubon's *Birds of America* 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

10.4. Tbl Commands

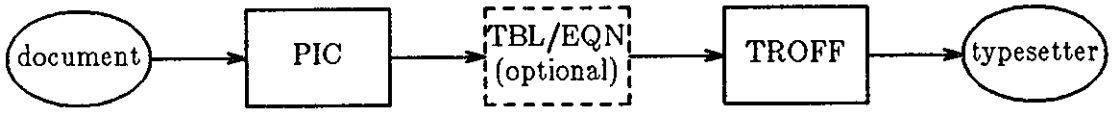
Table 10-1: *tbl* Command Characters and Words

Command	*Meaning*
`a A`	Alphabetic subcolumn
`allbox`	Draw box around all items
`b B`	Boldface item
`box`	Draw box around table
`c C`	Centered column
`center`	Center table in page
`doublebox`	Doubled box around table
`e E`	Equal width columns
`expand`	Make table full line width
`f F`	Font change
`i I`	Italic item
`l L`	Left adjusted column
`n N`	Numerical column
nnn	Column separation
`p P`	Point size change
`r R`	Right adjusted column
`s S`	Spanned item
`t T`	Vertical spanning at top
`tab (`*x*`)`	Change data separator character
`T{ T}`	Text block
`v V`	Vertical spacing change
`w W`	Minimum width value
`.`*xx*	Included *troff* command
|	Vertical line
||	Double vertical line
`^`	Vertical span
`\^`	Vertical span
`=`	Double horizontal line
`_`	Horizontal line
`\_`	Short horizontal line

Chapter 11

PIC — A Graphics Language for Typesetting

Pic is a language for describing how to draw simple figures on devices such as typesetters. *Pic* acts as a preprocessor to ***troff***. The basic objects in *Pic* are boxes, circles, ellipses, lines, arrows, arcs, spline curves, and text. These objects may be placed anywhere, at positions specified absolutely or in terms of previous objects. The example below illustrates the general capabilities of the language.



This picture was created with the input

```
ellipse "document"
arrow
box "PIC"
arrow
box "TBL/EQN" "(optional)" dashed
arrow
box "TROFF"
arrow
ellipse "typesetter"
```

Pic is another ***troff*** processor; it passes most of its input through untouched, but translates commands between `.PS` and `.PE` into ***troff*** commands that draw the pictures.

---

[6] **This User Manual was derived from a paper written by Brian W. Kernighan, Bell Laboratories, Murray Hill, New Jersey.**

11.1. Introduction

Pic is a language for drawing simple pictures. It operates as yet another ***troff***[1] preprocessor, (in the same style as ***eqn***[2], ***tbl***[3] and ***refer***[4]), with pictures marked by .PS and .PE.

Pic was inspired partly by Chris Van Wyk's early work on ***ideal***[5]; it has somewhat the same capabilities, but quite a different flavor. In particular, *Pic* is much more procedural—a picture is drawn by specifying (sometimes in painful detail) the motions that one goes through to draw it. Other direct influences include the ***picture*** language [6] and the V viewgraph language [7].

This paper is primarily a user's manual for ***pic***; a discussion of design issues and user experience may be found in [8].

The next section shows how to use *Pic* in the most simple way. Subsequent sections describe how to change the sizes of objects when the defaults are wrong, and how to change their positions when the standard positioning rules are wrong. An appendix describes the language succinctly and more or less precisely.

11.2. Basics

Pic provides boxes, lines, arrows, circles, ellipses, arcs, and splines (arbitrary smooth curves), plus facilities for positioning and labeling them. The picture below shows all of the fundamental objects (except for splines) in their default sizes:

---

[1] J. F. Ossanna, "NROFF/TROFF User's Manual," *UNIX Programmer's Manual*, vol. 2, Bell Laboratories, Murray Hill, N.J., January 1979. Section 22

[2] Brian W. Kernighan and Lorinda L. Cherry, "A System for Typesetting Mathematics," *Communications of the ACM*, vol. 18, no. 3, pp. 151-157, 1975.

[3] DNL, M. E. Lesk, "Tbl — A Program to Format Tables," *UNIX Programmer's Manual*, vol. 2, Bell Laboratories, Murray Hill, N.J., January 1979. Section 10

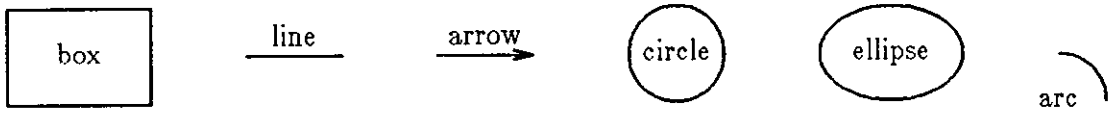
[4] DNL, M. E. Lesk, "Some Applications of Inverted Indexes on the UNIX System," *UNIX Programmer's Manual*, vol. 2, Bell Laboratories, Murray Hill, N.J., January 1979. Section 11

[5] Christopher J. Van Wyk and C. J. Van Wyk, "A Graphics Typesetting Language," *SIGPLAN Symposium on Text Manipulation*, Portland, Oregon, June, 1981.

[6] John C. Beatty, "PICTURE — A picture-drawing language for the Trix/Red Report Editor," Lawrence Livermore Laboratory Report UCID-30156, April 1977.

[7] Anon., "V — A viewgraph generating language," Bell Laboratories internal memorandum, May 1979.

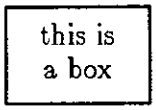
[8] B. W. Kernighan, "PIC — A Language for Typesetting Graphics," *Software Practice & Experience*, vol. 12, no. 1, pp. 1-21, January, 1982.



Each picture begins with `.PS` and ends with `.PE`; between them are commands to describe the picture. Each command is typed on a line by itself. For example

```
.PS
box "this is" "a box"
.PE
```

creates a standard box (¾ inch wide, ½ inch high) and centers the two pieces of text in it:



Each line of text is a separate quoted string. Quotes are mandatory, even if the text contains no blanks. Of course there needn't be any text at all. Each line is printed in the current size and font, centered horizontally, and separated vertically by the current ***troff*** line spacing.

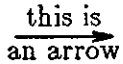
Pic does not center the drawing itself, but the default definitions of `.PS` and `.PE` in the `-ms` macro package do.

You can use `circle` or `ellipse` in place of `box`:

Text is centered on lines and arrows; if there is more than one line of text, the lines are centered above and below:

```
.PS
arrow "this is" "an arrow"
.PE
```

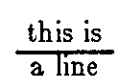
produces



and

```
line "this is" "a line"
```

gives



Boxes and lines may be dashed or dotted; just add the word **dashed** or **dotted** after **box** or **line**.

Arcs by default turn 90 degrees counterclockwise from the current direction; you can make them turn clockwise by saying **arc cw**. So

```
line; arc; arc cw; arrow
```

produces

A spline might well do this job better; we will return to that shortly.

As you might guess,

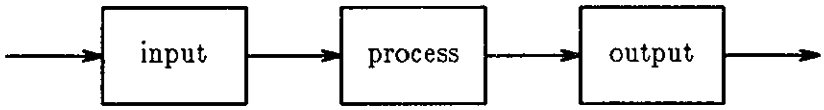
```
arc; arc; arc; arc
```

draws a circle, though not very efficiently.

Objects are normally drawn one after another, left to right, and connected at the obvious places. Thus the input

```
arrow; box "input"; arrow; box "process"; arrow; box "output"; arrow
```

produces the figure



If you want to leave a space at some place, use **move**:

```
box; move; box; move; box
```

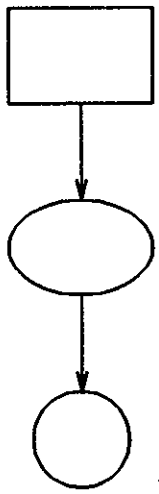
produces

Notice that several commands can be put on a single line if they are separated by semicolons.

Although objects are normally connected left to right, this can be changed. If you specify a direction (as a separate object), subsequent objects are joined in that direction. Thus

```
down; box; arrow; ellipse; arrow; circle
```

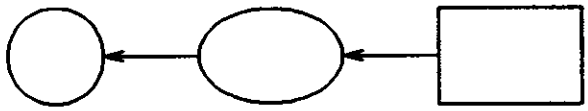
produces



and

```
left; box; arrow; ellipse; arrow; circle
```

produces



Each new picture begins going to the right.

Normally, figures are drawn at a fixed scale, with objects of a standard size. It is possible, however, to arrange that a figure be expanded to fit a particular width. If the .PS line contains a number, the drawing is forced to be that many inches wide, with the height scaled proportionately. Thus

```
.PS 3.5i
```

makes the picture 3.5 inches wide.

Pic is pretty dumb about the size of text in relation to the size of boxes, circles, and so on. There is as yet no way to say "make a box that just fits around this text" or "make this text fit inside this circle" or "draw a line as long as this text." All of these facilities are useful, so the limitations may go away in the fullness of time, but don't hold your breath. In the meantime, tight fitting of text can generally only be done by trial and error.

Speaking of errors, if you make a grammatical error in the way you describe a picture, *Pic* complains and try to indicate where. For example, the invalid input

```
box arrow box
```

draws the message

```
pic: syntax error near line 5, file -
 context is
        box arrow ^  box
```

The caret ^ marks the place where the error was first noted; it typically ***follows*** the word in error.

11.3. Controlling Sizes of Objects

This section deals with how to control the sizes of objects when the 'default' sizes are not what is wanted. The next section deals with positioning them when the default positions are not right.

Each object that *Pic* knows about (boxes, circles, etc.) has associated dimensions, like height, width, radius, and so on. By default, *Pic* tries to choose sensible default values for these dimensions, so that simple pictures can be drawn with a minimum of fuss and bother. All of the figures and motions shown so far have been in their default sizes:

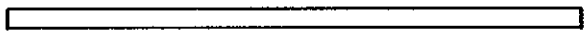
Table 11-1: PIC Objects and their Standard Sizes

Object	*Standard Size*
box	¾″ wide × ½″ high
circle	½″ diameter
ellipse	¾″ wide × ½″ high
arc	½″ radius
line or arrow	½″ long
move	½″ in the current direction

When necessary, you can make any object any size you want. For example, the input

```
box width 3i height 0.1i
```

draws a long, flat box



3 inches wide and 1/10 inch high. There must be no space between the number and the "`i`" that indicates a measurement in inches. In fact, the "`i`" is optional; all positions and dimensions are taken to be in inches.

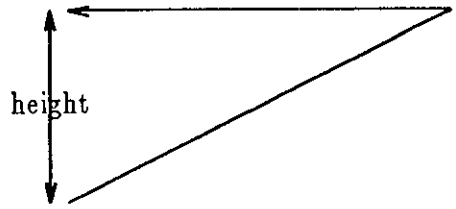
Giving an attribute like **`width`** changes only the one instance of the object. You can also change the default size for all objects of a particular type, as discussed later.

The attributes of **`height`** (which you can abbreviate to **`ht`**) and **`width`** (or **`wid`**) apply to boxes, circles, ellipses, and to the head on an arrow. The attributes of **`radius`** (or **`rad`**) and **`diameter`** (or **`diam`**) can be used for circles and arcs if they seem more natural.

Lines and arrows are most easily drawn by specifying the amount of motion from where one is right now, in terms of directions. Accordingly the words **`up`**, **`down`**, **`left`** and **`right`** and an optional distance can be attached to **`line`**, **`arrow`**, and **`move`**. For example,

```
.PS
line up 1i right 2i
arrow left 2i
move left 0.1i
line <-> down 1i "height"
.PE
```

draws

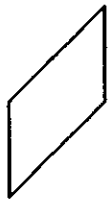


The notation `<->` indicates a two-headed arrow; use `->` for a head on the end and `<-` for one on the start. Lines and arrows are really the same thing; in fact, `arrow` is a synonym for `line ->`.

If you don't put any distance after `up`, `down`, etc., *Pic* uses the standard distance. So

```
line up right; line down; line down left; line up
```

draws the parallelogram



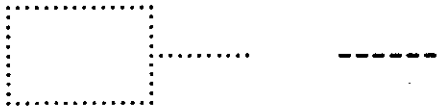
Warning: a very common error (which hints at a language defect) is to say

```
line 3i
```

A direction is needed:

```
line right 3i
```

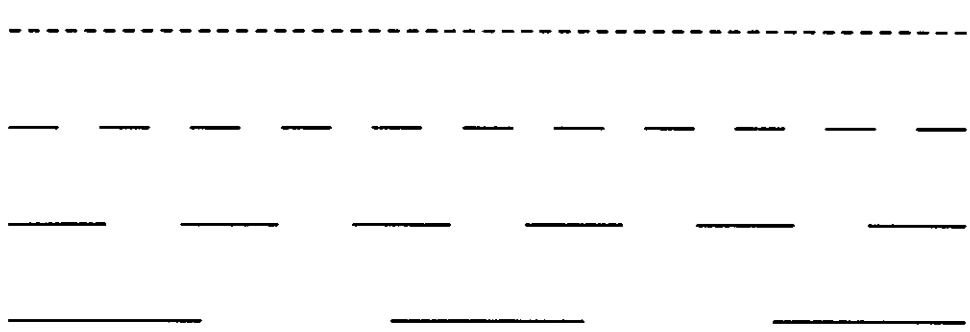
Boxes and lines may be dotted or dashed:



comes from

```
box dotted; line dotted; move; line dashed
```

If there is a number after **dot**, the dots are that far apart. You can also control the size of the dashes (at least somewhat): if there is a length after the word **dashed**, the dashes are that long, and the intervening spaces are as close as possible to that size. So, for instance,



comes from the inputs (as separate pictures)

```
line right 5i dashed
line right 5i dashed 0.25i
line right 5i dashed 0.5i
line right 5i dashed 1i
```

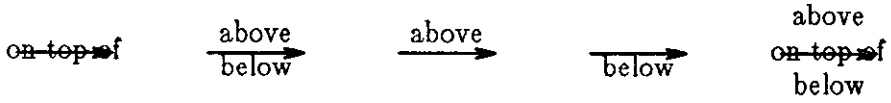
Sorry, but circles and arcs can't be dotted or dashed yet, and probably never will be.

You can make any object invisible by adding the word **invis(ible)** after it. This is particularly useful for positioning things correctly near text, as we will see later.

Text may be positioned on lines and arrows:

```
.PS
arrow "on top of"; move
arrow "above" "below"; move
arrow "above" above; move
arrow "below" below; move
arrow "above" "on top of" "below"
.PE
```

produces



The "width" of an arrowhead is the distance across its tail; the "height" is the distance along the shaft. The arrowheads in this picture are default size.

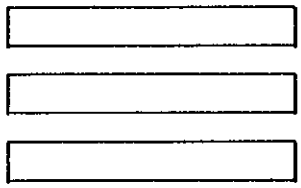
As we said earlier, arcs go 90 degrees counterclockwise from where you are right now, and **arc cw** changes this to clockwise. The default radius is the same as for circles, but you can change it with the **rad** attribute. It is also easy to draw arcs between specific places; this is described in the next section.

To put an arrowhead on an arc, use one of **<-**, **->** or **<->**.

In all cases, unless an explicit dimension for some object is specified, you get the default size. If you want an object to have the same size as the previous one of that kind, add the word **same**.

Thus in the set of boxes given by

```
down; box ht 0.2i wid 1.5i; move down 0.15i; box same; move same; box same
```



the dimensions set by the first **box** are used several times; similarly, the amount of motion for the second **move** is the same as for the first one.

11.3.1. Variables for Controlling Size of Objects

It is possible to change the default sizes of objects by assigning values to certain variables:

```
boxwid, boxht
linewid, lineht
dashwid
circlerad
arcrad
ellipsewid, ellipseht
movewid, moveht
arrowwid, arrowht        (These refer to the arrowhead.)
```

So if you want all your boxes to be long and skinny, and relatively close together,

```
boxwid = 0.1i; boxht = 1i
movewid = 0.2i
box; move; box; move; box
```

gives

Pic works internally in what it thinks are inches. Setting the variable **scale** to some value causes all dimensions to be scaled down by that value. Thus, for example, **scale=2.54** causes dimensions to be interpreted as centimeters.

The number given as a width in the **.PS** line overrides the dimensions given in the picture; this can be used to force a picture to a particular size even when coordinates have been given in inches. Experience indicates that the easiest way to get a picture of the right size is to enter its dimensions in inches, then if necessary add a width to the **.PS** line.

11.4. Controlling Positions of Objects

You can place things anywhere you want; *Pic* provides a variety of ways to talk about places. *Pic* uses a standard Cartesian coordinate system, so any point or object has an *x* and *y* position. The first object is placed with its start at position 0,0 by default. The *x,y* position of a box, circle or ellipse is its geometrical center; the position of a line or motion is its beginning; the position of an arc is the center of the corresponding circle.

Position modifiers like `from`, `to`, `by` and `at` are followed by an *x,y* pair, and can be attached to boxes, circles, lines, motions, and so on, to specify or modify a position.

You can also use `up`, `down`, `right`, and `left` with `line` and `move`. Thus

```
.PS 2
box ht 0.2 wid 0.2 at 0,0 "1"
move to 0.5,0           # or "move right 0.5"
box "2" same            # use same dimensions as last box
move same                       # use same motion as before
box "3" same
.PE
```

draws three boxes, like this:

Note the use of `same` to repeat the previous dimensions instead of reverting to the default values.

Comments can be used in pictures; they begin with a # and end at the end of the line.

Attributes like `ht` and `wid` and positions like `at` can be written out in any order. So

```
box ht 0.2 wid 0.2 at 0,0
box at 0,0 wid 0.2 ht 0.2
box ht 0.2 at 0,0 wid 0.2
```

are all equivalent, though the last is harder to read and thus less desirable.

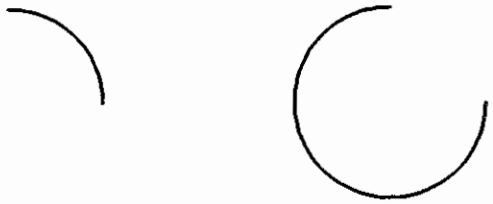
The `from` and `to` attributes are particularly useful with arcs, to specify the endpoints. By default, arcs are drawn counterclockwise,

```
arc from 0.5i,0 to 0,0.5i
```

is the short arc and

```
arc from 0,0.5i to 0.5i,0
```

is the long one:



If the `from` attribute is omitted, the arc starts where you are now and goes to the point given by `to`. The radius can be made large to provide flat arcs:

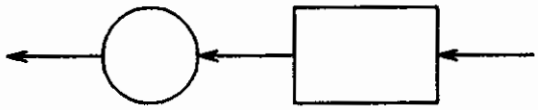
```
arc -> cw from 0,0 to 2i,0 rad 15i
```

produces

We said earlier that objects are normally connected left to right. This is an over-simplification. The truth is that objects are connected together in the direction specified by the most recent `up`, `down`, `left` or `right` (either alone or as part of some object). Thus, in

```
arrow left; box; arrow; circle; arrow
```

the `left` implies connection towards the left:



This could also be written as

```
left; arrow; box; arrow; circle; arrow
```

Objects are joined in the order determined by the last `up`, `down`, etc., with the entry point of the second object attached to the exit point of the first. Entry and exit points for boxes, circles and ellipses are on opposite sides, and the start and end of lines, motions and arcs. It's not entirely clear that this automatic connection and direction selection is the right design, but it seems to simplify many examples.

If a set of commands is enclosed in braces `{...}`, the current position and direction of motion when the group is finished is exactly where it was when entered. Nothing else is restored. There is also a more general way to group objects, using `[` and `]`, which is discussed in a later section.

11.5. Labels and Corners

Objects can be labelled or named so that you can talk about them later. For example,

```
.PS
Box1:
        box ...
        # ... other stuff ...
        move to Box1
.PE
```

Place names have to begin with an upper case letter (to distinguish them from variables, which begin with lower case letters). The name refers to the "center" of the object, which is the geometric center for most things. It's the beginning for lines and motions.

Other combinations also work:

```
line from Box1 to Box2
move to Box1 up 0.1 right 0.2
move to Box1 + 0.2,0.1  # same as previous
line to Box1 - 0.5,0
```

The reserved name `Here` may be used to record the current position of some object, for example as

```
Box1:  Here
```

Labels are variables — they can be reset several times in a single picture, so a line of the form

```
Box1:  Box1 + 1i,1i
```

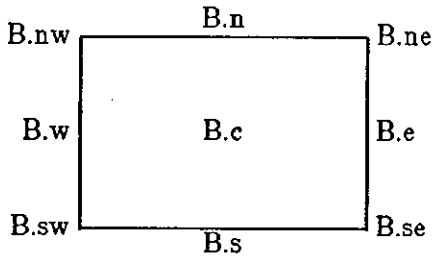
is perfectly legal.

You can also refer to previously drawn objects of each type, using the word `last`. For example, given the input

```
box "A"; circle "B"; box "C"
```

then '`last box`' refers to box `C`, '`last circle`' refers to circle `B`, and '`2nd last box`' refers to box `A`. Numbering of objects can also be done from the beginning, so boxes `A` and `C` are '`1st box`' and '`2nd box`' respectively.

To cut down the need for explicit coordinates, most objects have "corners" named by compass points:



The primary compass points may also be written as `.r`, `.b`, `.l`, and `.t`, for ***right***, ***bottom***, ***left***, and ***top***. The box above was produced with

```
.PS
B: box "B.c"
" B.e" at B.e ljust
" B.ne" at B.ne ljust
" B.se" at B.se ljust
"B.s" at B.s below
"B.n" at B.n above
"B.sw " at B.sw rjust
"B.w " at B.w rjust
"B.nw " at B.nw rjust
.PE
```

Note the use of `ljust`, `rjust`, `above`, and `below` to alter the default positioning of text, and of a blank with some strings to help space them away from a vertical line.

Lines and arrows have a `start`, an `end` and a `center` in addition to corners. Arcs have only a center, a start, and an end. There are a host of (that is, too many) ways to talk about the corners of an object. Besides the compass points, almost any sensible combination of `left`, `right`, `top`, `bottom`, `upper` and `lower` works. Furthermore, if you don't like the '.' notation, as in

```
last box.ne
```

you can instead say

```
upper right of last box
```

Prolixity like

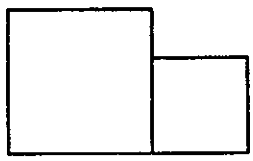
```
line from upper left of 2nd last box to bottom of 3rd last ellipse
```

begins to wear after a while, but it is descriptive. This part of the language is probably fat that will get trimmed.

It is sometimes easiest to position objects by positioning some part of one at some part of another, for example the northwest corner of one at the southeast corner of another. The `with` attribute in *Pic* permits this kind of positioning. For example,

```
box ht 0.75i wid 0.75i
box ht 0.5i wid 0.5i with .sw at last box.se
```

produces

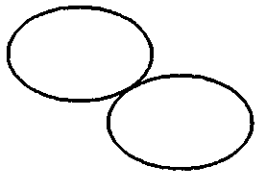


Notice that the corner after `with` is written `.sw`.

As another example, consider

```
ellipse; ellipse with .nw at last ellipse.se
```

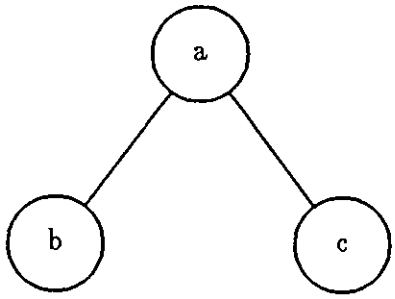
which makes



Sometimes it is desirable to have a line intersect a circle at a point which is not one of the eight compass points that *Pic* knows about. In such cases, the proper visual effect can be obtained by using the attribute `chop` to chop off part of the line:

```
circle "a"
circle "b" at 1st circle - (0.75i, 1i)
circle "c" at 1st circle + (0.75i, -1i)
line from 1st circle to 2nd circle chop
line from 1st circle to 3rd circle chop
```

produces



By default the line is chopped by `circlerad` at each end. This may be changed:

```
line ... chop r
```

chops both ends by ***r***, and

```
line ... chop r1 chop r2
```

chops the beginning by ***r1*** and the end by ***r2***.

There is one other form of positioning that is sometimes useful, to refer to a point some fraction of the way between two other points. This can be expressed in *Pic* as

```
fraction of the way between position1 and position2
```

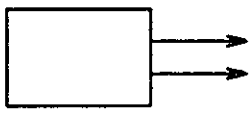
fraction is any expression, and ***position1*** and ***position2*** are any positions. You can abbreviate this rather windy phrase; 'of the way' is optional, and the whole thing can be written instead as

fraction < *position1* , *position2* >

As an example,

```
box
arrow right from 1/3 of the way between last box.ne and last box.se
arrow right from 2/3 <last box.ne, last box.se>
```

produces



Naturally, the distance given by ***fraction*** can be greater than 1 or less than 0.

11.6. Variables and Expressions

It's generally a bad idea to write everything in absolute coordinates if you are likely to change things. *Pic* variables let you parameterize your picture:

```
a = 0.5;  b = 1

box wid a ht b
ellipse wid a/2 ht 1.5*b
move to Box1 - (a/2, b/2)
```

Expressions may use the standard operators +, -, *, /, and %, and parentheses for grouping.

Probably the most important variables are the predefined ones for controlling the default sizes of objects, listed in Section 3. These may be set at any time in any picture, and retain their values until reset.

You can use the height, width, radius, and x and y coordinates of any object or corner in an expression:

```
Box1.x          # the x coordinate of Box1
Box1.ne.y               # the y coordinate of the northeast corner of Box1
Box1.wid                # the width of Box1
Box1.ht         # and its height
2nd last circle.rad     # the radius of the 2nd last circle
```

Any pair of expressions enclosed in parentheses defines a position; furthermore such positions can be added or subtracted to yield new positions:

```
( x , y )
( x1 , y1 ) + ( x2 , y2 )
```

If p_1 and p_2 are positions, then

(p_1 , p_2)

refers to the point

(p_1.x , p_2.y)

11.7. More on Text

Normally, text is centered at the geometric center of the object it is associated with. The attribute `ljust` causes the left end to be at the specified point (which means that the text lies to the right of the specified place!), and `rjust` puts the right end at the place. **above** and **below** center the text one half line space in the given direction.

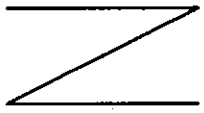
At the moment you can ***not compound text attributes: however natural it might seem, it is*** illegal to say `"..." above ljust`. This will be fixed eventually.

Text is most often an attribute of some other object, but you can also have self-standing text:

```
"this is some text" at 1,2 ljust
```

11.8. Lines and Splines

A "line" may actually be a path, that is, it may consist of connected segments like this:



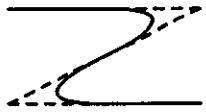
This line was produced by

```
line right 1i then down .5i left 1i then right 1i
```

A spline is a smooth curve guided by a set of straight lines just like the line above. It begins at the same place, ends at the same place, and in between is tangent to the mid-point of each guiding line. The syntax for a spline is identical to a (path) line except for using **spline** instead of `line`. Thus:

```
line dashed right 1i then down .5i left 1i then right 1i
spline from start of last line \
  right 1i then down .5i left 1i then right 1i
```

produces



Long input lines can be split by ending each piece with a backslash.

The elements of a path, whether for line or spline, are specified as a series of points, either in absolute terms or by **up**, **down**, etc. If necessary to disambiguate, the word **then** can be used to separate components, as in

```
spline right then up then left then up
```

which is not the same as

```
spline right up left up
```

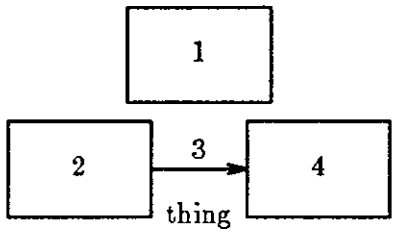
At the moment, arrowheads may only be put on the ends of a line or spline; splines may not be dotted or dashed.

11.9. Blocks

Any sequence of *Pic* statements may be enclosed in brackets `[...]` to form a block, which can then be treated as a single object, and manipulated rather like an ordinary box. For example, if we say

```
box "1"
[ box "2"; arrow "3" above; box "4" ] with .n at last box.s - (0,0.1)
"thing" at last [].s
```

we get



Notice that "last"-type constructs treat blocks as a unit and don't look inside for objects: "`last box.s`" refers to box 1, not box 2 or 4. You can use `last []`, etc., just like `last box`.

Blocks have the same compass corners as boxes (determined by the bounding box). It is also possible to position a block by placing either an absolute coordinate (like `0,0`) or an internal label (like `A`) at some external point, as in

```
[ ...; A: ...; ... ] with .A at ...
```

Blocks join with other things like boxes do (that is, at the center of the appropriate side). It's not clear that this is the right thing to do, so it may change.

Names of variables and places within a block are local to that block, and thus do not affect variables and places of the same name outside. You can get at the internal place names with constructs like

```
last [].A
```

or

```
B.A
```

where B is a name attached to a block like so:

```
B : [ ... ;  A: ...;  ]
```

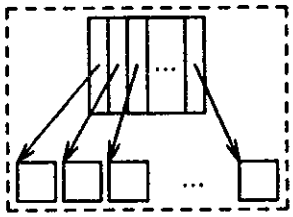
When combined with **define** statements (next section), blocks provide a reasonable simulation of a procedure mechanism.

Although blocks nest, it is currently possible to look only one level deep with constructs like B.A, although A may be further qualified (that is, B.A.sw or top of B.A are legal).

The following example illustrates most of the points made above about how blocks work:

```
h = .5i
dh = .02i
dw = .1i
[
        Ptr: [
                boxht = h; boxwid = dw
                A: box
                B: box
                C: box
                box wid 2*boxwid "..."
                D: box
        ]
        Block: [
                boxht = 2*dw; boxwid = 2*dw
                movewid = 2*dh
                A: box; move
                B: box; move
                C: box; move
                box invis "..." wid 2*boxwid; move
                D: box
        ] with .t at Ptr.s - (0,h/2)
        arrow from Ptr.A to Block.A.nw
        arrow from Ptr.B to Block.B.nw
        arrow from Ptr.C to Block.C.nw
        arrow from Ptr.D to Block.D.nw
]
box dashed ht last [].ht+dw wid last [].wid+dw at last []
```

This produces



11.10. Macros

Pic provides a rudimentary macro facility, the simple form of which is identical to that in ***eqn***:

```
define  name  X replacement text X
```

defines ***name*** to be the ***replacement text***; `X` is any character that does not appear in the replacement. Any subsequent occurrence of ***name*** is replaced by ***replacement text***.

Macros with arguments are also available. The replacement text of a macro definition may contain occurrences of `$1` through `$9`; these are replaced by the corresponding actual arguments when the macro is invoked. The invocation for a macro with arguments is

```
name(arg1, arg2, ...)
```

Non-existent arguments are replaced by null strings.

As an example, one might define a `square` by

```
define square X box ht $1 wid $1  $2 X
```

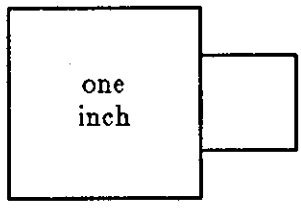
Then

```
square(1i, "one" "inch")
```

calls for a one inch square with the obvious label, and

```
square(0.5i)
```

calls for a square with no label:



Coordinates like x,y may be enclosed in parentheses, as in (x,y), so they can be included in a macro argument.

11.11. TROFF Interface

Pic is usually run as a ***troff*** preprocessor:

```
tutorial% pic -Tdevice file | iroff -mmacro_package
tutorial%
```

where ***device*** is the device you're formatting for. In the Sun installation, the device is coded as **-Timp** (imp for impress). Run ***pic*** before ***eqn*** and ***tbl*** if they are also present.

If the `.PS` line looks like

```
.PS <file
```

then the contents of `file` are inserted in place of the `.PS` line (whether or not the file contains `.PS` or `.PE`).

Other than this file inclusion facility, *Pic* copies the `.PS` and `.PE` lines from input to output intact, except that it adds two things right on the same line as the `.PS`:

```
.PS h w
```

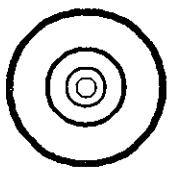
`h` and `w` are the picture height and width in units. The `-ms` macro package has simple definitions for `.PS` and `.PE` that cause pictures to be centered and offset a bit from surrounding text.

If "`.PF`" is used instead of `.PE`, the position after printing is restored to where it was before the picture started, instead of being at the bottom. ("F" is for "flyback.")

Any input line that begins with a period is assumed to be a ***troff*** command that makes sense at that point; it is copied to the output at that point in the document. It is asking for trouble to add spaces or in any way fiddle with the line spacing here, but point size and font changes are generally harmless. So, for example,

```
.ps 24
circle radius .4i at 0,0
.ps 12
circle radius .2i at 0,0
.ps 8
circle radius .1i at 0,0
.ps 6
circle radius .05i at 0,0
.ps 10  \" don't forget to restore size
```

gives



It is also safe to muck about with sizes and fonts and local motions within quoted strings ("...") in *pic*, as long as the changes made are changed back before exiting the string. For example, to print Italic text in 8-point type, use

```
ellipse "\s8\fISmile!\fP\s0"
```

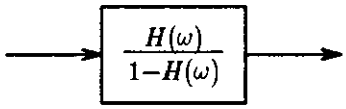
This produces

This is essentially the same rule as applies in *eqn*.

There is a subtle problem with complicated equations inside *Pic* pictures — they come out wrong if *eqn* has to leave extra vertical space for the equation. If your equation involves more than subscripts and superscripts, you must add to the beginning of each equation the extra information `space 0`:

```
arrow
box "$space 0 {H( omega )} over {1 - H( omega )}$"
arrow
```

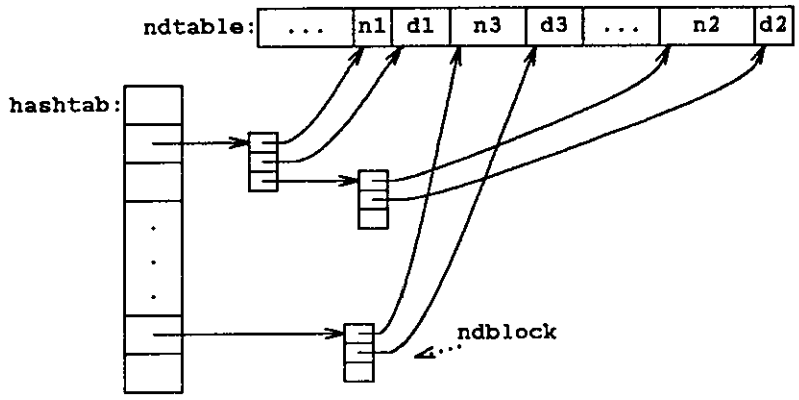
This produces



Pic normally generates commands for a new version of *troff* that has operators for drawing graphical objects like lines, circles, and so on. As distributed, *Pic* assumes that its output is going to the Mergenthaler Linotron 202 unless told otherwise with the `-T` option. At present, the other alternatives are `-Tcat` (the Graphic Systems CAT, which does slanted lines and curves badly) and `-Taps` (the Autologic APS-5). It is likely that the option will already have been set to the proper default for your system, unless you have a choice of typesetters.

11.12. Some Examples

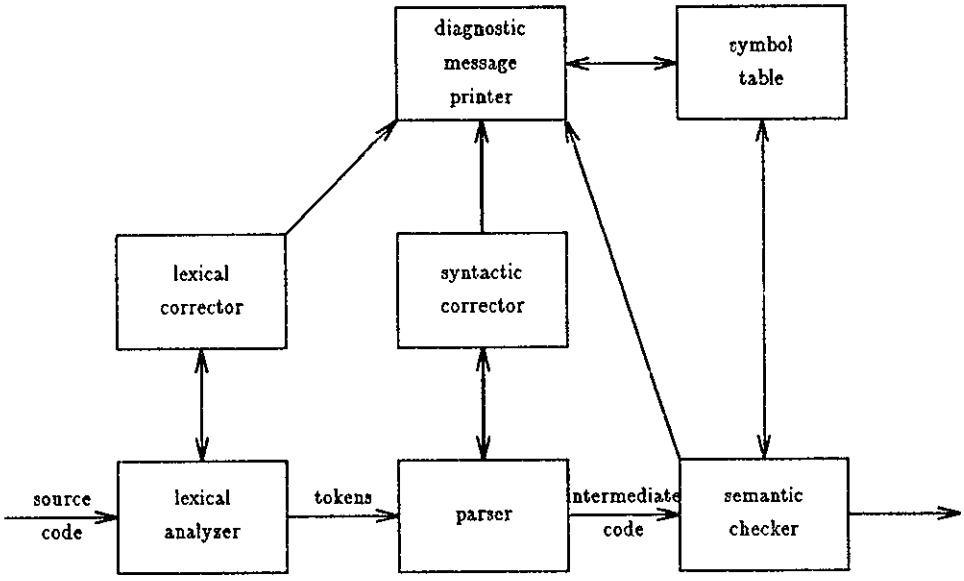
Herewith a handful of larger examples:



The input for the picture above was:

```
define ndblock X
        box wid boxwid/2 ht boxht/2
        down;  box same with .t at bottom of last box;   box same
X
boxht = .2i; boxwid = .3i; circlerad = .3i
down; box; box; box; box ht 3*boxht "." "." "."
L: box; box; box invis wid 2*boxwid "hashtab:" with .e at 1st box .w
right
Start: box wid .5i with .sw at 1st box.ne + (.4i,.2i) "..."
N1: box wid .2i "n1";  D1: box wid .3i "d1"
N3: box wid .4i "n3";  D3: box wid .3i "d3"
box wid .4i "..."
N2: box wid .5i "n2";  D2: box wid .2i "d2"
arrow right from 2nd box
ndblock
spline -> right .2i from 3rd last box then to N1.sw + (0.05i,0)
spline -> right .3i from 2nd last box then to D1.sw + (0.05i,0)
arrow right from last box
ndblock
spline -> right .2i from 3rd last box to N2.sw-(0.05i,.2i) to N2.sw+(0.05i,0)
spline -> right .3i from 2nd last box to D2.sw-(0.05i,.2i) to D2.sw+(0.05i,0)
arrow right 2*linewid from L
ndblock
spline -> right .2i from 3rd last box to N3.sw + (0.05i,0)
spline -> right .3i from 2nd last box to D3.sw + (0.05i,0)
circle invis "ndblock"  at last box.e + (.7i,.2i)
arrow dotted from last circle to last box chop
box invis wid 2*boxwid "ndtable:" with .e at Start.w
```

This is the second example:



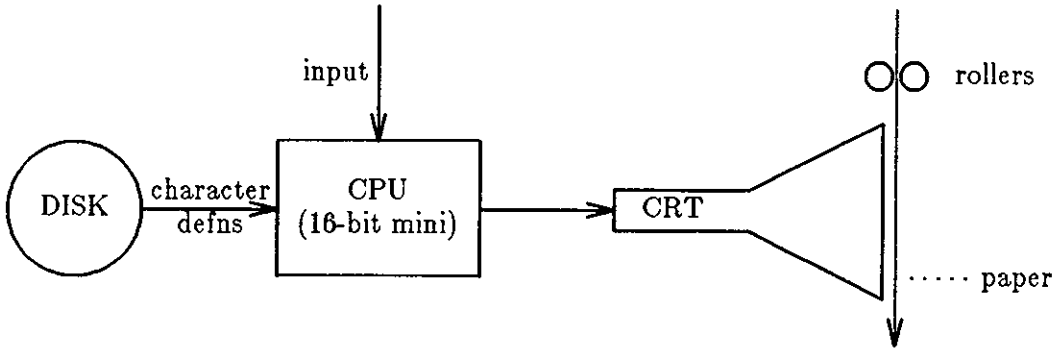
This is the input for the picture:

```
.PS 5
.ps 8
        arrow "source" "code"
LA:     box "lexical" "analyzer"
        arrow "tokens" above
P:      box "parser"
        arrow "intermediate" "code"
Sem:    box "semantic" "checker"
        arrow

        arrow <-> up from top of LA
LC:     box "lexical" "corrector"
        arrow <-> up from top of P
Syn:    box "syntactic" "corrector"
        arrow up
DMP:    box "diagnostic" "message" "printer"
        arrow <-> right  from right of DMP
ST:     box "symbol" "table"
        arrow from LC.ne to DMP.sw
        arrow from Sem.nw to DMP.se
        arrow <-> from Sem.top to ST.bot
.PE
```

There are eighteen objects (boxes and arrows) in the picture, and one line of *Pic* input for each; this seems like an acceptable level of verbosity.

The next example is the following:



Basic Digital Typesetter

This is the input for example 3:

```
.KS
.PS 5i
circle "DISK"
arrow "character" "defns"
box "CPU" "(16-bit mini)"
{ arrow <- from top of last box up "input " rjust }
arrow
CRT: "   CRT" ljust
line from CRT - 0,0.075 up 0.15 \
then right 0.5 \
then right 0.5 up 0.25 \
then down 0.5+0.15 \
then left 0.5 up 0.25 \
then left 0.5

Paper: CRT + 1.0+0.05,0
arrow from Paper + 0,0.75 to Paper - 0,0.5
{ move to start of last arrow down 0.25
  { move left 0.015; circle rad 0.05 }
  { move right 0.015; circle rad 0.05; "   rollers" ljust }
}
"  paper" ljust at end of last arrow right 0.25 up 0.25
line left 0.2 dotted
.PE
.ce
Basic Digital Typesetter
.sp
.KE
```

11.13. Final Observations

Pic is not a sophisticated tool. The fundamental approach — Cartesian coordinates and real measurements — is not the easiest thing in the world to work with, although it does have the merit of being in some sense sufficient. Much of the syntactic sugar (or corn syrup) — corners,

joining things implicitly, etc. — is aimed at making positioning and sizing automatic, or at least relative to previous things, rather than explicit.

Nonetheless, *Pic* does seem to offer some positive values. Most notably, it is integrated with the rest of the standard Unix document preparation software. In particular, it positions text correctly in relation to graphical objects; this is not true of any of the interactive graphical editors that I am aware of. It can even deal with equations in a natural manner, modulo the space 0 nonsense alluded to above.

A standard question is, "Wouldn't it be better if it were interactive?" The answer seems to be both yes and no. If one has a decent input device (which I do not), interaction is certainly better for sketching out a figure. But if one has only standard terminals (at home, for instance), then a linear representation of a figure is better. Furthermore, it is possible to generate *Pic* input from a program: I have used *awk* [9] to extract numbers from a report and generate the *Pic* commands to make histograms. This is hard to imagine with most of the interactive systems I know of.

In any case, the issue is far from settled; comments and suggestions are welcome.

11.13.1. Acknowledgements

I am indebted to Chris Van Wyk for ideas from several versions of ***ideal***. He and Doug McIlroy have also contributed algorithms for line and circle drawing, and made useful suggestions on the design of ***pic***. Theo Pavlidis contributed the basic spline algorithm. Charles Wetherell pointed out reference [2] to me, and made several valuable criticisms on an early draft of the language and manual. The exposition in this version has been greatly improved by suggestions from Jim Blinn. I am grateful to my early users — Brenda Baker, Dottie Luciani, and Paul Tukey — for their suggestions and cheerful use of an often shaky and clumsy system.

6

[9] A. V. Aho, P. J. Weinberger, and B. W. Kernighan, "AWK - A Pattern Scanning and Processing Language," *Software Practice and Experience*, vol. 9, pp. 267-280, April 1979.

11.14. PIC Reference Manual

11.14.1. Pictures

The top-level object in *Pic* is the "picture":

picture:
.PS *optional-width*
element-list
.PE

If ***optional-width*** is present, the picture is made that many inches wide, regardless of any dimensions used internally. The height is scaled in the same proportion.

If instead the line is

```
.PS <f
```

the file `f` is inserted in place of the `.PS` line.

If `.PF` is used instead of `.PE`, the position after printing is restored to what it was upon entry.

11.14.2. Elements

An ***element-list*** is a list of elements (what else?); the elements are

element:
primitive attribute-list
placename : *element*
placename : *position*
variable = *expression*
direction
troff-command
{ *element-list* }
[*element-list*]

Elements in a list must be separated by newlines or semicolons; a long element may be continued by ending the line with a backslash. Comments are introduced by a # and terminated by a newline.

Variable names begin with a lower case letter; place names begin with upper case. Place and variable names retain their values from one picture to the next.

The current position and direction of motion are saved upon entry to a `{...}` block and restored upon exit.

Elements within a block enclosed in `[...]` are treated as a unit; the dimensions are determined by the extreme points of the contained objects. Names, variables, and direction of motion within a block are local to that block.

troff-command is any line that begins with a period. Such lines are assumed to make sense in the context where they appear; accordingly, if it doesn't work, don't call.

11.14.3. Primitives

The primitive objects are

primitive:
```
box
circle
ellipse
arc
line
arrow
move
spline
"any text at all"
```

`arrow` is a synonym for `line ->`.

11.14.4. Attributes

An ***attribute-list*** is a sequence of zero or more attributes; each attribute consists of a keyword, perhaps followed by a value. In the following, *e* is an expression and ***opt-e*** an optional expression.

attribute:

`h(eigh)t` *e*	`wid(th)` *e*
`rad(ius)` *e*	`diam(eter)` *e*
`up` *opt-e*	`down` *opt-e*
`right` *opt-e*	`left` *opt-e*
`from` *position*	`to` *position*
`at` *position*	`with` *corner*
`by` *e, e*	`then`
`dotted` *opt-e*	`dashed` *opt-e*
`chop` *opt-e*	`->` `<-` `<->`
`same`	`invis`
text-list	

Missing attributes and values are filled in from defaults. Not all attributes make sense for all primitives; irrelevant ones are silently ignored. These are the currently meaningful attributes:

```
box:
        height, width, at, dotted, dashed, invis, same, text
circle and ellipse:
        radius, diameter, height, width, at, invis, same, text
arc:
        up, down, left, right, height, width, from, to, at, radius,
        invis, same, cw, <-, ->, <->, text
line, arrow
        up, down, left, right, height, width, from, to, by, then,
        dotted, dashed, invis, same, <-, ->, <->, text
spline:
        up, down, left, right, height, width, from, to, by, then,
        invis, <-, ->, <->, text
move:
        up, down, left, right, to, by, same, text
"text...":
        at, text
```

The attribute `at` implies placing the geometrical center at the specified place. For lines, splines and arcs, `height` and `width` refer to arrowhead size.

11.14.5. Text

Text is normally an attribute of some primitive; by default it is placed at the geometrical center of the object. Stand-alone text is also permitted. A ***text-list*** is a list of text items; a text item is a quoted string optionally followed by a positioning request:

```
text-item:
        "..."
        "..." center
        "..." ljust
        "..." rjust
        "..." above
        "..." below
```

If there are multiple text items for some primitive, they are centered vertically except as qualified. Positioning requests apply to each item independently.

Text items can contain ***troff*** commands for size and font changes, local motions, etc., but make sure that these are balanced so that the entering state is restored before exiting.

11.14.6. Positions and places

A position is ultimately an ***x,y*** coordinate pair, but it may be specified in other ways.

position:

e, e
place ± *e, e*
(*position, position* **)**
e [**of the way**] **between** *position* **and** *position*
e **<** *position , position* **>**

The pair *e, e* may be enclosed in parentheses.

place:

placename *optional-corner*
corner **placename**
Here
corner **of** *nth primitive*
nth primitive optional-corner

A ***corner*** is one of the eight compass points or the center or the start or end of a primitive. (Not text!)

corner:

.n .e .w .s .ne .se .nw .sw
.t .b .r .l
.c .start .end

Each object in a picture has an ordinal number; ***nth*** refers to this.

nth:

*n***th**
*n***th last**

Since barbarisms like **1th** are barbaric, synonyms like **1st** and **3st** are accepted as well.

11.14.7. Variables

The built-in variables and their default values are:

```
boxwid 0.75i            boxht 0.5i
circlerad 0.25i
ellipsewid 0.75i        ellipseht 0.5i
arcrad 0.25i
linewid 0.5i            lineht 0.5i
movewid 0.5i            movewid 0.5i
arrowht 0.1i            arrowwid 0.05i
dashwid 0.1i
scale 1
```

These may be changed at any time, and the new values remain in force until changed again. Dimensions are divided by **scale** during output.

11.14.8. Expressions

Expressions in *Pic* are evaluated in floating point. All numbers representing dimensions are taken to be in inches.

expression:

```
e + e
e - e
e * e
e / e
e % e  (modulus)
- e
( e )
variable
number
place .x
place .y
place .ht
place .wid
place .rad
```

11.14.9. Definitions

The `define` statement is not part of the grammar.

`define`:

```
define name X replacement text X
```

Occurrences of `$1` through `$9` in the replacement text is replaced by the corresponding arguments if `name` is invoked as

```
name(arg1, arg2, ...)
```

Non-existent arguments are replaced by null strings. ***Replacement text*** may contain newlines.

Chapter 12

Typesetting Mathematics with eqn

This chapter[7] explains how to use the *eqn* preprocessor for printing mathematics on a phototypesetter and provides numerous examples after which to model equations in your documents.

You describe mathematical expressions in an English-like language that the *eqn* program translates into *troff* commands for final *troff* formatting. In other words, *eqn* sets the mathematics while *troff* does the body of the text. *Eqn* provides accurate and relatively easy mathematical phototypesetting, which is not easy to accomplish with normal typesetting machines. Because the mathematical expressions are imbedded in the running text of a manuscript, the entire document is produced in one process. For example, you can set in-line expressions like $\lim_{x \to \pi/2} (\tan x)^{\sin 2x} = 1$ or display equations like

$$G(z) = e^{\ln G(z)} = \exp\left(\sum_{k \geq 1} \frac{S_k z^k}{k}\right) = \prod_{k \geq 1} e^{S_k z^k / k}$$

$$= \left(1 + S_1 z + \frac{S_1^2 z^2}{2!} + \cdots\right)\left(1 + \frac{S_2 z^2}{2} + \frac{S_2^2 z^4}{2^2 \cdot 2!} + \cdots\right) \cdots$$

$$= \sum_{m \geq 0} \left(\sum_{\substack{k_1, k_2, \ldots, k_m \geq 0 \\ k_1 + 2k_2 + \cdots + mk_m = m}} \frac{S_1^{k_1}}{1^{k_1} k_1!} \frac{S_2^{k_2}}{2^{k_2} k_2!} \cdots \frac{S_m^{k_m}}{m^{k_m} k_m!} \right) z^m$$

Eqn knows relatively little about mathematics. In particular, mathematical symbols like +, −, ×, parentheses, and so on have no special meanings. *Eqn* is quite happy to set these symbols, and they will look good.

Eqn also produces mathematics with *nroff*. The input is identical, but you have to use the programs *neqn* instead of *eqn* and *troff*. Of course, some things won't look as good because your workstation or terminal does not provide the variety of characters, sizes and fonts that a phototypesetter does, but the output is usually adequate for proofreading.

12.1. Displaying Equations — '.EQ' and '.EN'

To tell *eqn* where a mathematical expression begins and ends, mark it with lines beginning '.EQ' and '.EN'. Thus if you type the lines:

[7] The material in this chapter is derived from *A System for Typesetting Mathematics*, B.W. Kernighan, L. L. Cherry and *Typesetting Mathematics — User's Guide*, B.W. Kernighan, L.L. Cherry, Bell Laboratories, Murray Hill, New Jersey.

```
.EQ
x=y+z
.EN
```

your output will look like:

$$x=y+z$$

Eqn copies '.EQ' and '.EN' through untouched. This means that you have to take care of things like centering, numbering, and so on yourself. The common way is to use the ***troff*** and ***nroff*** macro package package '**−ms**', which provides macros for centering, indenting, left-justifying and making numbered equations.

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'.

You can also supplement ***eqn*** with ***troff*** commands as desired; for example, you can produce a centered display with the input:

```
.ce
.EQ
x sub i = y sub i ...
.EN
```

which produces

$$x_i=y_i \cdots$$

You can call out any of these 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 you can enter in-line expressions like π_i^2 without '.EQ ' and '.EN'. This is described in ***Shorthand for In-line Equations.***

12.2. Running eqn and neqn

To print a document that contains mathematics on the phototypesetter, use:

```
logo% eqn files | troff −options | lpr −t −Printer
logo%
```

Troff or your installation's equivalent does the formatting, which is sent to your phototypesetter as indicated by **−P***rinter*. If you use the **−ms** macro package for example, type:

```
logo% eqn files | troff −ms −t | lpr −t −Printer
logo%
```

To display equations on the standard output, your workstation screen, use ***nroff*** as follows:

logo% **neqn** *files* ¦ **nroff** −*options*

The language for equations recognized by ***neqn*** is identical to that of ***eqn***, although of course the output is more restricted. You can use the online rendition of the mathematical formulae for proofing, but the output does not accurately represent the symbols and fonts. You can of course pipe the output through ***more*** for easier viewing:

logo% **neqn** *files* ¦ **nroff** −*options* ¦**more**

or redirect it to a file:

logo% **neqn** *files* ¦ **nroff** −*options* >*newfile*

To use a GSI or DASI terminal as the output device, type:

logo% **neqn** *files* ¦ **nroff** −**T***x*

where is the terminal type you are using, such as ***300*** or ***300S***. To send ***neqn*** output to the printer, type:

logo% **neqn** *file* ¦ **nroff** −*options* ¦ **lpr** −**P***printer*

You can use ***eqn*** and ***neqn*** with the ***tbl*** program for setting tables that contain mathematics. Use ***tbl*** before ***eqn*** or ***neqn***, like this:

logo% **tbl** *files* ¦ **eqn** ¦ **troff** −*options*
logo%

or

logo% **tbl** *files* ¦ **neqn** ¦ **nroff** −*options*

12.3. Putting Spaces in the Input Text

Eqn throws away spaces and newlines within an expression and leaves normal text alone. Thus between '.EQ' and '.EN ',

```
.EQ
x=y+z
.EN
```

and

```
.EQ
x = y + z
.EN
```

and

```
.EQ
x  =  y
   + z
.EN
```

all produce the same output:

$$x=y+z$$

You should 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 they are often hard to fix if you make a mistake.

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. To do this, surround a special word by ordinary spaces (or tabs or newlines), as shown in the previous section.

You can also make special words stand out by surrounding them with tildes or circumflexes:

```
.EQ
x~=~2~pi~int~sin~(~omega~t~)~dt
.EN
```

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:

$$x = 2 \pi \int \sin (\omega t) dt$$

You can also use braces '{ }' and double quotes ' "..." ' to separate special words; these characters which have special meanings are described later.

Remembering that a blank is a delimiter can be a problem. For instance, a common mistake is typing:

```
.EQ
f(x sub i)
.EN
```

which produces

$$f(x_{i)}$$

instead of

$$f(x_i)$$

Eqn cannot tell that the right parenthesis is not part of the subscript. Type instead:

```
.EQ
f(x sub i )
.EN
```

12.4. Producing Spaces in the Output Text

To force extra spaces into the *output,* use a tilde '~' for each space you want:

```
.EQ
x~=~y~+~z
.EN
```

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. Use tabs to position pieces of an expression, but you must use *troff* commands to set the tab stops.

12.5. Symbols, Special Names, and Greek Letters

Eqn knows some mathematical symbols, some mathematical names, and the Greek alphabet. For example,

```
.EQ
x=2 pi int sin ( omega t)dt
.EN
```

produces

$$x=2\pi\int\sin(\omega t)dt$$

Here the spaces in the input are **necessary** to 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 ***very*** common error is to type ***f(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 $f(pi)$ instead of $f(\pi)$.

A complete list of ***eqn*** names appears in ***Precedences and Keywords.*** You can also use special characters available in ***troff*** for anything ***eqn*** doesn't know about.

12.6. Subscripts and Superscripts — 'sub' and 'sup'

To obtain subscripts and superscripts, use the words ***sub*** and ***sup.***

```
.EQ
x sup 2 + y sub k
.EN
```

gives

$$x^2+y_k$$

Eqn takes care of all the size changes and vertical motions needed to make the output look right. You must surround the words ***sub*** and ***sup*** by spaces; ***x sub2*** gives you $xsub2$ instead of x_2. As another example, consider:

```
.EQ
x sup 2 + y sup 2 = z sup 2
.EN
```

which produces:

$$x^2+y^2=z^2$$

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

```
.EQ
y = (x sup 2)+1
.EN
```

which causes

$$y=(x^{2)+1}$$

instead of the intended

$$y=(x^2)+1$$

which is produced by:

```
.EQ
y = (x sup 2 )+1
.EN
```

Subscripted subscripts and superscripted superscripts also work:

```
.EQ
x sub i sub 1
.EN
```

is

$$x_{i_1}$$

A subscript and superscript on the same thing are printed one above the other if the subscript comes ***first:***

```
.EQ
x sub i sup 2
.EN
```

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 $x^y{}_z$.

12.7. Grouping Equation Parts — '{' and '}'

Normally, the end of a subscript or superscript is marked simply by a blank, tab, tilde, and so on. If the subscript or superscript is something that has to be typed with blanks in it, use the braces '{' and '}' to mark the beginning and end of the subscript or superscript:

```
.EQ
e sup {i omega t}
.EN
```

is

$$e^{i\omega t}$$

You can ***always*** use braces to force ***eqn*** to treat something as a unit, or just to make your intent perfectly clear. Thus:

```
.EQ
x sub {i sub 1} sup 2
.EN
```

is

$$x_{i_1}^2$$

with braces, but

```
.EQ
x sub i sub 1 sup 2
.EN
```

is

$$x_{i_1^2}$$

which is rather different.

Braces can occur within braces if necessary:

```
.EQ
e sup {i pi sup {rho +1}}
.EN
```

is

$$e^{i\pi^{\rho+1}}$$

The general rule is that anywhere you could use some single entry like x, you can use an arbitrarily complicated entry if you enclose it in braces. ***Eqn*** looks 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 causes ***eqn*** to complain bitterly.

Occasionally you have to print braces. To do this, enclose them in double quotes, like ' "{" '. Quoting is discussed in more detail in ***Quoted Text.***

12.8. Fractions — 'over'

To make a fraction, use the word ***over:***

```
.EQ
a+b over 2c =1
.EN
```

gives

$$\frac{a+b}{2c}=1$$

The line is made the right length and positioned automatically.

```
.EQ
a+b over c+d+e = 1
.EN
```

produces

$$\frac{a+b}{c+d+e}=1$$

Use braces to clarify what goes over what:

```
.EQ
{alpha + beta} over {sin (x)}
.EN
```

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

```
.EQ
-b sup 2 over pi
.EN
```

is $\frac{-b^2}{\pi}$ instead of $-b^{\frac{2}{\pi}}$ The rules which decide which operation is done first in cases like this are summarized in ***Precedences and Keywords.*** When in doubt, however, use ***braces*** to make clear what goes with what.

12.9. Square Roots — 'sqrt'

To draw a square root, use ***sqrt:***

```
.EQ
sqrt a+b
.EN
```

produces

$$\sqrt{a+b}$$

and

```
.EQ
sqrt a+b + 1 over sqrt {ax sup 2 +bx+c}
.EN
```

is

$$\sqrt{a+b}+\frac{1}{\sqrt{ax^2+bx+c}}$$

Note: Square roots of tall quantities look sloppy because a root-sign big enough to cover the quantity is too dark and heavy:

```
.EQ
sqrt {a sup 2 over b sub 2}
.EN
```

is

$$\sqrt{\frac{a^2}{b_2}}$$

Big square roots are generally better written as something to a power:

$$(a^2/b_2)^{\frac{1}{2}}$$

which is

```
.EQ
(a sup 2 /b sub 2 ) sup {1 over 2}
.EN
```

12.10. Summation, Integral, and Other Large Operators

To produce summations, integrals, and similar constructions, use:

```
.EQ
sum from i=0 to {i= inf} x sub i
.EN
```

which produces

$$\sum_{i=0}^{i=\infty} x_i$$

Notice that you use braces to indicate where the upper part $i=\infty$ begins and ends. No braces are necessary for the lower part $i=0$, because it does not contain any 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:

```
.EQ
int  prod  union  inter
.EN
```

become, respectively,

$$\int \quad \prod \quad \bigcup \quad \bigcap$$

Since the thing before the ***from*** can be anything, even something in braces, ***from-to*** can often be used in unexpected ways:

```
.EQ
lim from {n -> inf} x sub n =0
.EN
```

is

$$\lim_{n \to \infty} x_n = 0$$

12.11. Size and Font Changes

By default, equations are set in 10-point type with standard mathematical conventions to determine what characters are in roman and what in italic. Although ***eqn*** makes a valiant attempt to use aesthetically 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 thing

that follows them; they revert to the normal situation at the end of it. Thus

```
.EQ
bold x y
.EN
```

is

$$\mathbf{x}y$$

and

```
.EQ
size 14 bold x = y +
  size 14 {alpha + beta}
.EN
```

gives

$$\mathbf{x}=y+\alpha+\beta$$

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

```
.EQ
size 12 { ... }
.EN
```

Legal sizes which may follow ***size*** are the same as those allowed in ***troff***: 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. This is easier because you don't have to know what the current size is.

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 good 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 severe nuisance to have to write out a size and font change for each equation. Accordingly, you can set a 'global' size or font which thereafter affects all equations. 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 thereafter. 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: you can change the global font and size as often as needed. For example, in a footnote[8] you will typically want the size of equations to match the size of the footnote

[8] Like this one, in which we have a few random expressions like x_i and π^2. The sizes for these were set

text, which is two points smaller than the main text. Don't forget to reset the global size at the end of the footnote.

12.12. Diacritical Marks

To get funny 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 ***under*** are made the right length for the entire construct, as in $\overline{x+y+z}$; other marks are centered. For example

```
.EQ
x dot under + x hat + y tilde
+ X hat + Y dotdot = z+Z bar
.EN
```

produces

$$\dot{\underline{x}}+\hat{x}+\tilde{y}+X+\ddot{Y}=\overline{z+Z}$$

12.13. Quoted Text

Any input entirely within quotes ("...") is not subject to any of the font changes and spacing adjustments that you normally set. This provides a way to do your own spacing and adjusting if needed:

```
.EQ
italic "sin(x)" + sin (x)
.EN
```

is

$$sin(x)+\sin(x)$$

You also use quotes to get braces and other ***eqn*** keywords printed:

```
.EQ
"{ size alpha }"
.EN
```

is

$$\{\ size\ alpha\ \}$$

by the command *gsize ~−2.*

and

```
.EQ
roman "{ size alpha }"
.EN
```

is

$$\{ \text{size alpha} \}$$

The construction ' "" ' is often used as a place-holder when grammatically ***eqn*** needs something, but you don't actually want anything in your output. For example, to make ^{2}He, you can't just type ***sup 2 roman He*** because a ***sup*** has to be a superscript ***on*** something. Thus you must say

```
.EQ
"" sup 2 roman He
.EN
```

To get a literal quote use '\"'. ***Troff*** characters like ***\(bs*** can appear unquoted, but more complicated things like horizontal and vertical motions with ***\h*** and ***\v*** should always be quoted.

12.14. Lining Up Equations — 'mark' and 'lineup'

Sometimes it's necessary to line up a series of equations at some horizontal position, often at an equals sign. To do this, use the 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

$$x+y=z$$

$$x=1$$

For reasons out of the scope of this chapter, when you use ***eqn*** and '**−ms**', use either '.EQ I' or '.EQ L', as ***mark*** and ***lineup*** don't work with centered equations. Also bear in mind that ***mark*** doesn't look ahead;

```
.EQ
x mark =1
...
x+y lineup =z
.EN
```

isn't going to work, because there isn't room for the ***x+y*** part after the ***mark*** has processed the ***x***.

12.15. Big Brackets

To get big brackets '[ ]', braces '{ }', parentheses '( )', and bars '| |' around things, use the ***left*** and ***right*** commands:

```
.EQ
left { a over b + 1 right }
~=~ left ( c over d right )
+ left [ e right ]
.EN
```

is

$$\left\{\frac{a}{b}+1\right\} = \left(\frac{c}{d}\right) + \left[e\right]$$

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:

```
.EQ
left floor x over y right floor
<= left ceiling a over b right ceiling
.EN
```

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. Second, big left and right parentheses often look poor, because the character set is poorly designed.

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.

12.16. Piles — 'pile'

There is a general facility for making vertical piles of things; it comes in several flavors. For example:

```
.EQ
A ~=~ left [
  pile { a above b above c }
  ~~ pile { x above y above z }
right ]
.EN
```

will make

$$A = \begin{bmatrix} a & x \\ b & y \\ c & z \end{bmatrix}$$

The elements of the pile are centered one above another at the right height for most purposes. There can be as many elements as you want. The keyword ***above*** is used to separate the pieces; put braces 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. For example:

```
.EQ
roman sign (x)~=~
left {
   lpile {1 above 0 above -1}
   ~~ lpile
    {if~x>0 above if~x=0 above if~x<0}
.EN
```

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.

12.17. Matrices — 'matrix'

It is also possible to make matrices. For example, to make a neat array like

$$x_i \quad x^2$$
$$y_i \quad y^2$$

you have to type

```
.EQ
matrix {
  ccol { x sub i above y sub i }
  ccol { x sup 2 above y sup 2 }
}
.EN
```

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.

A word of warning about matrices: ***each column must have the same number of elements in it***. Otherwise, results are unpredictable.

12.18. Shorthand for In-line Equations — 'delim'

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 you need variable names like x to be in italics. Although you can do this 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 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 $alpha sub i$ be the primary variable, and let $beta$ be zero.  Then we can show
that $x sub 1$ 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 $sum from i=1 to n x sub i$ does not interfere with the lines surrounding it.

The printed result looks like: Let α_i be the primary variable, and let β be zero. Then we can show that x_1 is ≥ 0.

To turn off the delimiters, use:

```
.EQ
delim off
.EN
```

Note: Don't use braces, tildes, circumflexes, or double quotes as delimiters; chaos will result.

12.19. Definitions — 'define'

Eqn provides a string-naming 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

```
.EQ
x sub i sub 1 + y sub i sub 1
.EN
```

appears repeatedly throughout a paper, you can save re-typing it each time by defining it like this:

```
.EQ
define  xy  'x sub i sub 1 + y sub i sub 1'
.EN
```

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 sure 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  xi1  ' xi sub 1 '
.EN
```

don't define something in terms of itself. A favorite error is to say

```
.EQ
define  X  ' roman X '
.EN
```

This is a guaranteed disaster, since X ***is*** now defined in terms of itself. If you say

```
.EQ
define  X  ' roman "X" '
.EN
```

however, the quotes protect the second X, and everything works fine.

You can redefine ***eqn*** keywords. You can make '/' mean ***over*** by saying

```
.EQ
define  /  ' over '
.EN
```

or redefine ***over*** as '/' with

```
.EQ
define  over  ' / '
.EN
```

If you need things to print on a workstation or terminal as well as on the phototypesetter, it is sometimes worth defining a symbol differently in ***neqn*** and ***eqn***. To do this, use ***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***.

12.20. Tuning the Spacing

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. You can get small extra horizontal spaces with tilde and circumflex. You can also say ***back n*** and ***fwd n*** to move small amounts horizontally. The ***n*** is how far to move in 1/100's of an em (an em is about the width of the letter 'm'.) Thus ***back 50*** moves back about half the width of an m. 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 anything if it is enclosed in braces.

12.21. Troubleshooting

If you make a mistake in an equation, like leaving out a brace, having one too many, or having a ***sup*** with nothing before it, ***eqn*** tells you with the message:

syntax error between lines x and y, file z

where ***x*** and ***y*** are approximately the lines between which the trouble occurred, and ***z*** is the name of the file in question. The line numbers are approximate; 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 nonexistent file.

If you want to check a document before actually printing it, run:

logo% **eqn** ***files*** **>/dev/null**

to throw away the output but display the messages.

If you use something like dollar signs as delimiters, it is easy to leave one out. You may also occasionally forget one half of a pair of macros or have an unbalanced font change. These can cause problems, but you can check for balanced pairs of delimiters and macros with ***checkeq*** and ***checknr***. For instance, to run ***checkeq*** on this chapter called ***eqn.ug*** to check for unbalanced pairs of '.EQ' and '.EN', type:

```
logo% checkeq eqn.ug
eqn.ug:
  New delims , line 2
   in EQ, line 2
  Spurious EN, line 46
  Delim off, line 1254
  New delims , line 1278
  New delims , line 1635
   in EQ, line 1635
  New delims ##, line 1991
  Delim off, line 1999
logo%
```

We left out the '.EQ' before the '.EN' on line 46 to show you some sample output. This also reports on the delimiters. You can also use ***checknr*** with specific options to check specifically for a particular macro pair. For example, to run ***checknr*** to check that there is an '.EQ' for every '.EN', type:

```
logo% checknr -s -f -a.EQ.EN eqn.ug
46: Unmatched .EN
logo%
```

Specify the macro pair you want to check for with the **-a** option and the six characters in the pair. The **-s** option ignores size changes and the *-f* option ignores font changes. See the user's manual on ***checknr*** for more details.

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, that is, offset from the body of the text with '.EQ' and '.EN', 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 '.EQ ... .EN' sequence. ***Eqn*** does warn about equations that are too long to fit on one line.

12.22. Precedences and Keywords

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
```

The operations that group to the left are:

```
over sqrt left right
```

All others group to the right. For example, in the expression

```
.EQ
a sup 2 over b
.EN
```

sup is defined to have a higher precedence than *over*, so this construction is parsed as $\frac{a^2}{b}$ instead of $a^{\frac{2}{b}}$. Naturally, you can always force a particular parsing by placing braces around expressions.

Digits, parentheses, brackets, punctuation marks, and the following 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

The following character sequences are recognized and translated as shown.

Table 12-1: Character Sequence Transalation

You Type	Translation
>=	$\geq$
<=	$\leq$
==	$\equiv$
!=	$\neq$
+-	$\pm$
->	$\rightarrow$
<-	$\leftarrow$
<<	$\ll$
>>	$\gg$
inf	∞
partial	∂
prime	$'$
approx	$\approx$
nothing	
cdot	$\cdot$
times	$\times$
del	∇
grad	∇
...	$\ldots$
,....,	$,\ldots,$
sum	$\sum$
int	$\int$
prod	$\prod$
union	$\cup$
inter	$\cap$

To obtain Greek letters, simply spell them out in whatever case you want:

Table 12-2: Greek Letters

You Type	Translation	You Type	Translation
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	o
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		

The *eqn* keywords, except for characters with names, follow.

Table 12-3: *eqn* Keywords

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	

12.23. Several Examples

Here is the complete source for several examples and for the three display equations in the introduction to this chapter.

Squareroot

Input:

```
.EQ
x = {-b + - sqrt{b sup 2-4ac}} over 2a
.EN
```

Output:

$$x = \frac{-b + - \sqrt{b^{2-4ac}}}{2a}$$

Summation, Integral, and Other Large Operators

Input:

```
.EQ
lim from {x -> pi /2} ( tan~x) = inf
.EN
```

Output:

$$\lim_{x \to \pi/2} (\tan\ x) = \infty$$

Input:

```
.EQ
sum from i=0 to infinity x sub i = pi over 2
.EN
```

Output:

$$\sum_{i=0}^{\infty} x_i = \frac{\pi}{2}$$

Input:

```
.EQ
lim from {x-> pi /2} ( tan~x) sup{sin~2x}~=~1
.EN
```

Output:

$$\lim_{x \to \pi/2} (\tan\ x)^{\sin 2x} = 1$$

Input:

```
.EQ
define emx "{e sup mx}"
define mab "{m sqrt ab}"
define sa "{sqrt a}"
define sb "{sqrt b}"
int dx over {a emx - be sup -mx}~=~
left { lpile {
    1 over {2 mab} ~log~
        {sa emx - sb}over{sa emx + sb}
above
 1 over mab~tanh sup -1 ( sa over sb emx )
above
 -1 over mab~coth sup-1 (sa over sb emx )
}
.EN
```

Output:

$$\int \frac{dx}{ae^{mx}-be^{-mx}} = \begin{cases} \frac{1}{2m\sqrt{ab}} \log \frac{\sqrt{a}e^{mx}-\sqrt{b}}{\sqrt{a}e^{mx}+\sqrt{b}} \\ \frac{1}{m\sqrt{ab}} \tanh^{-1}\left(\frac{\sqrt{a}}{\sqrt{b}}e^{mx}\right) \\ \frac{-1}{m\sqrt{ab}} \operatorname{coth} sup -1 \frac{(sa}{\sqrt{b}}e^{mx}) \end{cases}$$

Quoted Text

Input:

```
.EQ
lim~ roman "sup" ~x sub n = 0
.EN
```

Ouput:

$$\lim \sup x_n = 0$$

Big Brackets

Input:

```
.EQ
left [ x+y over 2a right ]~=~1
.EN
```

Output:

$$\left[\frac{x+y}{2a}\right] = 1$$

Fractions

Input:

```
.EQ
a sub 0 + b sub 1 over
 {a sub 1 + b sub 2 over
  {a sub 2 + b sub 3 over
   {a sub 3 + ...}}}
.EN
```

Output:

$$a_0+\cfrac{b_1}{a_1+\cfrac{b_2}{a_2+\cfrac{b_3}{a_3+\cdots}}}$$

Input:

```
.EQ I
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
```

Output:

$$G(z) = e^{\ln G(z)} = \exp\left(\sum_{k\geq 1}\frac{S_k z^k}{k}\right) = \prod_{k\geq 1} e^{S_k z^k/k}$$

Input:

```
.EQ I
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
```

Output:

$$= \left(1+S_1 z+\frac{S_1^2 z^2}{2!}+\cdots\right)\left(1+\frac{S_2 z^2}{2}+\frac{S_2^2 z^4}{2^2\cdot 2!}+\cdots\right)\cdots$$

Input:

```
.EQ I
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
```

Output:

$$= \sum_{m \geq 0} \left(\sum_{\substack{k_1, k_2 \ldots, k_m \geq 0 \\ k_1 + 2k_2 + \ldots + mk_m = m}} \frac{S_1^{k_1}}{1^{k_1} k_1!} \frac{S_2^{k_2}}{2^{k_2} k_2!} \cdots \frac{S_m^{k_m}}{m^{k_m} k_m!} \right) z^m$$

Shorthand for In-line Equations

Input:

```
.EQ
delim ##
.EN
```

Let #x sub i#, #y# and #alpha# be positive

Output:

Let x_i, y and α be positive

Chapter 13

Refer — A Bibliography System

13.1. Introduction

Refer is a bibliography system that supports data entry, indexing, retrieval, sorting, runoff, convenient citations, and footnote or endnote numbering. You can enter new bibliographic data into the database, index the selected data, and retrieve bibliographic references from the database. This document assumes you know how to use a Unix editor, and that you are familiar with the ***nroff*** and ***troff*** text formatters.

The **refer** program is a preprocessor for ***nroff*** and ***troff***, and works like like **eqn** and **tbl**. **Refer** is used for literature citations, rather than for equations and tables. Given incomplete but sufficiently precise citations, **refer** finds references in a bibliographic database. The complete references are formatted as footnotes, numbered, and placed either at the bottom of the page, or at the end of a chapter.

A number of related programs make **refer** easier to use. The **addbib** program is for creating and extending the bibliographic database; **sortbib** sorts the bibliography by author and date, or other selected criteria; and **roffbib** runs off the entire database, formatting it not as footnotes, but as a bibliography or annotated bibliography.

Once a full bibliography has been created, access time can be improved by making an index to the references with **indxbib**. Then, the **lookbib** program can be used to quickly retrieve individual citations or groups of citations. Creating this inverted index will speed up **refer,** and **lookbib** will allow you to verify that a citation is sufficiently precise to deliver just one reference.

13.2. Features

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. This field contains ***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*** and ***troff*** macros, and, with a few exceptions, the **refer** program itself does not pay attention to the labels. This implies that you can change the label codes, if you also change the macros used by ***nroff*** and ***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 (lines) should be supplied. Except for %A, the author field, each field should be given only once. In the case of multiple authors, the senior author should be entered first. Your entry in such a case, might look like this:

```
%A Brian W. Kernighan
%A P. J. Plauger
%T Software Tools in Pascal
%I Addison-Wesley
%C Reading, Massachusetts
%D 1981
```

The %Q is for organizational authors, or authors with Japanese or Arabic names, in which cases there is no clear last name. 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 an 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 \fBrefer\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 every time the reference is used. 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.

13.3. Data Entry with Addbib

The **addbib** program is for creating and extending bibliographic databases. You must give it the filename of your bibliography:

```
hostname% 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.

13.4. Printing the Bibliography

Sortbib is for sorting the bibliography by author (%A) and date (%D), or by data in other fields. Sortbib is quite useful for producing bibliographies and annotated bibliographies, which are seldom entered in strict alphabetical order.

Sortbib 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 **−s***KEYS* 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:

```
hostname% 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:

```
hostname% sortbib database | roffbib -Txerox -s1
```

will make accent marks work on a Xerox 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: N, V, L, and O. These 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.)

13.5. 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, 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 lineprinter, 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:

```
hostname% refer -p database paper | nroff -ms
hostname% refer -p database paper | tbl | nroff -ms
hostname% 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.

8

[10] Mike E. Lesk, "Some Applications of Inverted Indexes on the Unix System," in *Unix Programmer's Manual*, Bell Laboratories, Murray Hill, NJ, 1978.

13.6. 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 **−s***KEYS* 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 the **−s***KEYS* flag, since **−s** implies **−e**. See the section ***Printing the Bibliography*** for additional features of the **−s***KEYS* flag.

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 **−a***n* 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 first and last 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.

13.7. 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:

```
hostname% 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.

```
hostname% 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 \fBrefer\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:

```
hostname% lookbib /usr/dict/papers/Ind > kern.refs
> kernighan
> <CTRL-D>EOT
hostname% 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.

13.8. Refer Bugs and Some Solutions

13.8.1. Blanks at Ends of Lines

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 *ex* editor command —

```
g/ *$/s///
```

— or similar method to remove trailing blanks from your bibliography.

13.8.2. Interpolated Strings

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.

13.8.3. Interpreting Foreign Surnames

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*** and ***troff*** request meaning to insert a digit-width space. Because the second argument to the %A field contains no blank spaces to confuse the **refer** program, **refer** will treat the second field as a single word. This protects against faulty name reversal, and also against mis-sorting.

13.8.4. Footnote Numbers

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 use the following method immediately before the citation:

```
\fIet al.\fR
.[
awk aho kernighan weinberger
.]
```

Now the reference will be to Aho *et al.*[11] The **\fI** changes to italics, and the **\fR** changes back to Roman font. Both these requests are ***nroff*** and ***troff*** requests, not part of **−ms**. If and when a footnote number is added after this sequence, it will indeed appear in the output.

8

[11] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, *Awk — A Pattern Scanning and Text Processing Language,* Bell Laboratories, Murray Hill, NJ.

13.9. 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 Lesk 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/ms.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
%J	1	Journal Article
%B	3	Article in Book
1G	4	Report, Government Report
%I	2	Book
%M	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
%I 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 is best to redefine the **[.** and **.]** strings.

13.10. 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.

The **refer** macros for ***nroff*** and ***troff*** supplied by the **−ms** macro package reside in /usr/lib/ms/ms.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:

```
hostname% 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

```
hostname% 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 "ms.ref":

```
hostname% cp /usr/lib/ms/ms.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.

Chapter 14

Formatting Documents with the —me Macros

This chapter[9] describes the *—me* macro package text processing facility. The first part of each section presents the material in user's guide format and the second part lists the macro requests for quick reference. The chapter contents include descriptions of the basic requests, displays, annotations, such as footnotes, and how to use *—me* with ***nroff*** and ***troff***.

We assume that you are somewhat familiar with ***nroff*** and ***troff*** and that you know something about breaks, fonts, point sizes, the use and definition of number registers and strings, and scaling factors for ens, points, vertical line spaces (v's), etc. If you are a newcomer, try out the basic features as you read along.

All request names in *—me* follow a naming convention. You may define number registers, strings, and macros, provided that you use single-character, upper-case names or double character names consisting of letters and digits with at least one upper-case letter. Do not use special characters in the names you define. The word ***argument*** in this chapter means a word or number which appears on the same line as a request and which modifies the meaning of that request. Default parameter values are given in brackets. For example, the request

```
.sp
```

spaces one line, and

```
.sp 4
```

spaces four lines. The number '4' is an ***argument*** to the '.sp' request; it modifies '.sp' to produce four lines instead of one. Spaces separate arguments from the request and from each other.

14.1. Using —me

When you have your raw text ready, run the ***nroff*** formatter with the *—me* option to send the output to the standard output, your workstation screen. Type:

```
logo% 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 you omit the **—T** flag, a 'lowest common denominator' terminal is assumed; this is good for previewing output on most terminals.

For easier viewing, pipe the output to ***more*** or redirect it to another file.

For formatting on the phototypesetter with ***troff*** (or your installation's equivalent), use:

---

[9] The material in this chapter is derived from *Writing Papers with 'nroff' Using '—me'*, E.P. Allman and *'—me' Reference Manual*, E.P. Allman, University of California, Berkeley.

logo% **troff −me** *file*

14.2. Basic −me Requests

The following sections provide descriptions and examples of the basic *−me* requests.

14.2.1. Paragraphs

The *−me* package has requests for formatting standard, left block, and indented paragraphs.

14.2.1.1. Standard Paragraph — '.pp'

Begin standard paragraphs 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

Now is the time for all good men to come to the aid of their party. Four score and seven years ago,...

that is, a blank line followed by an indented first line.

Do not begin the sentences of a paragraph with a space, since blank lines and lines beginning with spaces cause a break. For example, if you type:

```
.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 is:

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 begins with a space character.

Because the first call to one of the paragraph macros defined in a section or the '.H' macro (see ***Section Headings***) ***initializes*** the macro processor, do not use any of the following requests: '.sc', '.lo', '.th', or '.ac'. Also, avoid changing parameters, notably page length and header and footer margins, which have a global effect on the format of the page.

14.2.1.2. Left Block Paragraphs — '.lp'

A formatted paragraph can start with a blank line and with the first line indented. You can get left-justified block-style paragraphs as shown throughout this manual by using '.lp' (left paragraph) instead of '.pp'.

14.2.1.3. Indented Paragraphs — '.ip' and '.np'

Sometimes you want to use paragraphs that have the ***body*** indented, and the first line exdented, that is, the opposite of indented, with a label. Use the '.ip' request for this. A word specified on the same line as '.ip' is printed in the margin, and the body is lined up at a specified position. 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 an '.ip' request, 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', type:

```
.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' begins a new line after the label. For example, the input:

```
.ip longlabel
This paragraph has 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.
```

produces:

longlabel
This paragraph has 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 oth-

er.

You can change the size of the label by using a second argument which is the size of the label. For example, you can produce the above example correctly by saying:

.ip longlabel 10

which will make the paragraph indent 10 spaces for this paragraph only. For example:

longlabel
This paragraph has 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.

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 1i

somewhere before the first call to '.ip'.

If you use '.ip' without an argument, no hanging tag is 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 does not have a 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 does not have a 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 '.H' 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.
.lp
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.

14.2.1.4. Paragraph Reference

.lp Begin left-justified paragraph. Centering and underlining are turned off if they were on, the font is set to **\n(pf** [1], the type size is set to **\n(pp** [10p], and a **\(nps** 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 **\n($1** [0] plus **\n(po** [0] unless the paragraph is inside a display (see '.ba' in ***Miscellaneous Requests***). At least the first two lines of the paragraph are kept together on a page.

.pp Like '.lp', except that it puts **\n(pi** [5n] 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 **\n(ii** [5n] spaces if *I* is not specified) more than a non-indented paragraph is (such as with '.lp'). The title *T* is exdented. 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' starts a new line.

.np An '.ip' variant that numbers paragraphs. Numbering is reset after an '.lp', '.pp', or '.H'. The current paragraph number is in **\n$p**.

14.3. Headers and Footers — '.he' and '.fo'

You can put arbitrary headers and footers 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. The first character of ***title*** (whatever it may be) is used as a delimiter to separate these three parts. You can use any character but avoid the backslash and double quote marks. The percent sign is replaced by the current page number whenever it is 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.

If there are two blanks adjacent anywhere in the title or more than eight blanks total, you must enclose three-part titles in single quotes.

Headers and footers are set in font **\n(tf** [3] and size **\n(tp** [10p]. Each of the definitions applies as of the ***next*** page.

Three number registers control the spacing of headers and footers. **\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). You can also specify the space between the top of the page and the header, the header and the first line of text, the bottom of the text and the footer, and the footer and the bottom of the page with the macros '.m1', '.m2', '.m3', and '.m4'.

14.3.1. Headers and Footers Reference

.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. Must be followed by a '.bp' or the end of input.

.$h Called at every page to print the header. May be redefined to provide fancy headers, such as, multi-line, 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 processing the header, initial saved floating keeps, etc.; in other words, this macro is called immediately before printing text on a page. Used for column headings and the like.

14.3.2. Double Spacing — '.ls 2'

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'.

14.3.3. Page Layout

You can change the way the printed copy looks, sometimes called the ***layout*** of the output page with the following requests. Most of these requests adjust the placing of 'white space' (blank lines or spaces). In these explanations, replace characters in italics with values you wish to use; bold characters represent characters which you should actually type.

Use '.bp' (break page) to start a new page.

The request '.sp *N*' leaves *N* lines of blank space. You can omit *N* to skip a single line or you can use the form '*N* i' (for *N* inches) or '*N* c' (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*' (indent) request changes the amount of white space on the left of the page. 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 '*N* i' or *N* c' 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 1i
.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.

You can center text lines with the '.ce' (center) 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, to 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' (break).

14.3.4. Underlining — '.ul'

Use the '.ul' (underline) request to underline text. The '.ul' request operates on the next input line when it is processed. You can underline multiple lines by stating a count of ***input*** lines to underline, followed by those lines, the same as with the '.ce' request. For example, the input:

```
.ul 2
The quick brown fox
jumped over the lazy dog.
```

underlines those words in ***nroff***. In ***troff*** they are italicized.

14.3.5. Displays

Use displays to set off sections of text from the body of the paper. Major quotes, tables, and figures are types of displays, as are all the examples used in this manual. All displays except centered text blocks are single spaced.

14.3.5.1. Major Quotes — '.(q' and '.)q'

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. Use '.(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,...

14.3.5.2. Lists — '.(l' and '.)l'

A ***list*** is an indented, single-spaced, unfilled display. You should use lists when the material to be printed should not be filled and justified like normal text. This is useful for columns of figures, for example. Surround the list text 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
```

to produce:

Alternatives to avoid deadlock are:

Lock in a specified order
Detect deadlock and back out one process
Lock all resources needed before proceeding

14.3.5.3. Keeps — '.(b' and '.)b', '.(z' and '.)z'

A ***keep*** is a display of lines which are kept on a single page if possible. Keeps are useful for printing diagrams, for example. 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 enough 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 a ***floating keep.***

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'. An example of a floating keep is:

```
.(z
.hl
Text of keep to be floated.
.sp
.ce
Figure 1. Example of a Floating Keep.
.hl
.)z
```

The '.hl' request draws a horizontal line so that the figure stands out from the text.

14.4. Fancy 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 produced by '.(l' is indented from both margins. For example, the input:

```
.(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 formatted as:

> 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!

Lists and blocks are also normally indented, while floating keeps are normally left justified. To get a left-justified list, type '.(l L'. To center a list line-for-line, type '.(l C'. For example, to get a filled, left-justified list, use:

```
.(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 you may 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' keep argument.

Centered blocks are ***not*** keeps, and you may use them 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

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 centers the centered block within the entire line rather than within the line minus the indent. Also, you must nest the center requests ***inside*** the keep requests.

14.4.1. Display Reference

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* [**I**] is **I** the list is indented by **\n(bi** [4n]; if it is **M**, the list is indented to the left margin; if it 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 it is **C**, the list is centered on a line-by-line basis. The list is set in font **\n(df** [0]. You must use a matching '.)l' to end the list. This macro is almost like '.DS' except that no attempt is made to keep the display on one page.

.)l — End list.

.(q — Begin major quote. The lines are single-spaced, filled, moved in from the main body of 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**, that is, one point smaller than the 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 is begun, 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**, when they are filled. The 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].

.)b End block.

.(z *m f* Begin floating keep. Like '.(b' except that the keep is ***floated*** 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.

14.4.2. 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 explicitly called for.

14.4.3. Footnotes — '.(f' and '.)f'

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 number.[1] The number is automatically incremented after every footnote. For example, the input:

```
.(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]

---

[1] Like this.

Make sure that the footnote appears ***inside*** the quote, so that the footnote will appear on the same page as the quote.

14.4.4. Delayed Text

Delayed text is very similar to a footnote except that it is printed when explicitly called for. Use this feature to put a list of references 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 refer to them with special characters* rather than numbers.

14.4.5. Indexes — '.(x' '.)x' and '.xp'

An ***index*** resembles delayed text, in that it is saved until called for. It is actually more like a table of contents, since the entries are not sorted alphabetically. 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'. An argument to the '.)x' indicates 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 specifies an explicitly null page number.

The '.xp' request prints the index.

For example, the input:

```
.(x
Sealing wax
.)x 9
.(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 9
.xp
```

generates:

---

9 [2]James R. Ware, *The Best of Confucius,* Halcyon House, 1950. Page 77.

9 *Such as an asterisk.

Sealing wax 9

Cabbages and kings

< etc. >

The '.(x' request may have a single character argument, specifying the ***name*** of the index; the normal index is **x**. Thus, you can maintain several ***indicies*** simultaneously, such as a list of tables and a table of contents.

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); you may have to rearrange the pages after printing.

14.4.6. Annotations Reference

.(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. You might use this at the end of each chapter.

.(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-columned 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 generate the footnote separator. You may redefine this macro to give other size lines or other types of separators. It currently draws a 1.5-inch line.

.(x *x* — Begin index entry. Index entries are saved in the index *x* until called up with '.xp'. Each entry is preceded by a **\n(xs** [0.2v] space. Each entry is 'undented' 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 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 are printed.

.xp *x* — Print index *x* [**x**]. The index is formated in the font, size, and so forth in effect at the time it is printed, rather than at the time it is collected.

14.5. Fancy 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 those used in this manual, and multicolumn output.

14.5.1. Section Headings — '.sh' and '.uh'

You can automatically generate section numbers, using the '.sh' request. You must tell '.sh' the ***depth*** of the section number and a section title. The depth specifies how many numbers separated by decimal points are to appear in the section number. For example, the section number '4.2.5' has a depth of three.

Section numbers are incremented if you add a number. Hence, you increase the depth, and 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 beginning section number 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 be numbered relative to this number.

There are more complex features which indent each section proportionally to the depth of the section. For example, if you type:

```
.nr si Nx
```

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
```

The request indents sections by one-half inch per level of depth in the section number. As another example, consider:

```
.nr si 3n
```

which gives three spaces of indent per section depth.

You can produce section headers without automatically generated numbers using:

.uh "Title"

which will do a section heading, but will not put a number on the section.

14.5.1.1. Section Heading Reference

.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**. If **\n(si** is non-zero, the base indent is set to **\n(si** times the section depth, and the section title is exdented (see '.ba'in *Miscellaneous Requests)*. 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. A '.sh' insures that there is enough room to print the section head plus the beginning of a paragraph, which is about 3 lines total. If you specify *a* through *f*, the section number is set to that number rather than incremented automatically. If any of *a* through *f* are a hyphen that number is not reset. If *T* is a single underscore (_), the section depth and numbering is reset, but the base indent is not reset and nothing is printed. 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 for '.sh'.

.$p *T B N* Print section heading. May be redefined to get fancier headings. *T* is the title passed 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, '.uh' passes only the first, and '.sx' passes three, but the first two are null strings. Be careful if you redefine this macro, as it is quite complex and subtle.

.$0 *T B N* Callled automatically after every call to '.$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, and *N* is the section depth.

.$1 – .$6 Traps called just before printing that depth section. May be defined to give variable spacing before sections. These macros are called from '.$p', so if you redefine that macro you may lose this feature.

14.5.2. Parts of the Standard 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
A BENCHMARK FOR THE NEW SYSTEM
.sp
by
.sp
J. P. Hacker
.)l
.bp
```

The request '.th' sets up the environment of the ***nroff*** processor to do a thesis. It defines the correct headers, footers, a page number in the upper right-hand corner only, sets the margins correctly, and double spaces.

Use the '.+c *T*' request 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"
```

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 standard PhD thesis.

If the title parameter *T* is omitted from the '.+c' request, the result is a chapter with no heading. You can also use this at the beginning of a paper.

Although papers traditionally have the abstract, table of contents, and so forth at the front, 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. At the end of the paper, give the '.++ P' request, which begins the preliminary part of the paper. After using 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. You may use '.+c' repeatedly to begin different parts of the front material for example, the abstract, the table of contents, acknowledgments, list of illustrations, and so on. You may also use the request '.++ B' to begin the bibliographic section at the end of the paper. For example, the paper might appear as outlined below. (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
A BENCHMARK FOR THE NEW SYSTEM
.sp
by
.sp
J.P. Hacker
.)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
```

Outline of a Sample Paper

14.5.2.1. Standard Paper Reference

.tp Begin title page. Spacing at the top of the page can occur, and headers and footers are suppressed. Also, the page number is not incremented for this page.

.th Set thesis mode. This defines the modes acceptable for a doctoral dissertation. 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. Use '.++' and '.+c' with it. This macro must be stated before initialization, that is, before the first call of a para-

graph macro or '.H'.

.++ *m H*

This request defines the section of the paper which you are typing. The section type is defined by *m*: 'C' means that you are entering the chapter portion of the paper, 'A' means that you are entering the appendix portion of the paper, 'P' means that the material following should be the preliminary portion (abstract, table of contents, etc.) portion of the paper, 'AB' means that you are entering the abstract (numbered independently from 1 in Arabic numerals), and 'B' means that you are entering the bibliographic portion at the end of the paper. You can also use the variants 'RC' and 'RA', which specify renumbering of pages from one 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 appendixes 'A.1' etc., type **.++ RA ''' \\\\n(ch.%'**. Precede each section (chapter, appendix, etc.) by the '.+c' request. When using ***troff***, it is easier to put the front material at the end of the paper, so that the table of contents can be collected and generated; you can then physically move this material to the beginning of the paper.

.+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 papers without a title page proper. '.$c' calls '.$C' as a hook so that chapter titles can be inserted into a table of contents automatically. The footnote numbering is reset to one.

.$c *T* Print chapter number (from **\n(ch)** and *T*. You can redefine this macro to your liking. It is defined by default to be acceptable for a standard PhD thesis. This macro calls '$C', which can be defined to make index entries, or whatever.

.$C *K N T*

This macro is called by '.$c'. It is normally undefined, but can be used to automatically insert index entries, or whatever. *K* is a keyword, either 'Chapter' or 'Appendix' (depending on the '.++' 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 Associattion for Computing Machines (ACM). 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*. Additionally, this macro loads the file ***/usr/lib/me/acm.me***, which may later be augmented with other macros for printing papers for ACM conferences. Note that this macro will not work correctly in ***troff***, since it sets the page length wider than the physical width of the phototypesetter roll.

14.5.3. Two-Column Output — '.2c'

You can get two column output automatically by using the request '.2c'. This produces everything after it 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'.

14.5.3.1. Columned Output Reference

.2c +*S N*

Enter two-column mode. The column separation is set to +*S* [4n, 0.5i in ACM mode] (saved in **\n($s**). The column width, calculated to fill the single column line length with both columns, is stored in **\n($l**. The current column is in **\n($c**. You can test register **\n($m** [1] to see if 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.

14.5.4. Defining Macros — '.de'

A ***macro*** is a collection of requests and text which you may use 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, type:

```
.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 command names in —*me*, always use upper-case letters as names. Avoid the names 'TS', 'TH', 'TE', 'EQ', and 'EN'.

14.5.5. 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 use something like:

```
.(z
.(c
Text of figure
.)c
.ce
Figure 5.
\!.(x f
\!Figure 5
\!.)x
.)z
```

which will give you a figure with a label and an entry in the index 'f', presumably a list of figures index. Because the index entry is read and interpreted when the keep is read, and not when it is printed, you have is to use the magic string ' \!' at the beginning of all the lines dealing with the index. Otherwise, the page number in the index is likely to be wrong. This defers index processing until the figure is generated, and guarantees that the page number in the index is correct. The same comments apply to blocks with '.(b' and '.)b'.

14.6. Using 'troff' for Phototypsetting

You can prepare documents for either displaying on a workstation or for phototypesetting using the ***troff*** formatting program.

14.6.1. 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 for use with the ***eqn*** and ***neqn*** mathematical equation processors. 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, 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 'l' does not overlap the quote sign in ***troff***.

There are also several ***pseudo-fonts*** available. For example, the input:

.u underlined

generates

underlined

and

.bx "words in a box"

produces

words in a box

You can also get bold italics with

.bi "bold italics"

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 give the requests:

.bi "some bold italics"
and
.bx "words in a box"

in the middle of a line, ***troff*** would overwrite the first and the box lines on the second would be poorly drawn.

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'; you can join input lines together without a space between them. For example, the input:

.u under \c
.i italics

generates under***italics*** , but if you type:

.u under
.i italics

the result is under ***italics*** as two words.

14.6.2. Point Sizes — '.sz'

The phototypesetter supports different sizes of type, measured in points. The default point size is 10 points for most text and eight points for footnotes. To change the point size, type:

.sz +*N*

where *N* is the size wanted in points. The 'vertical spacing,' that is, the distance between the bottom of most letters (the ***baseline***) and the adjacent line is set to be proportional to the type size.

Note: Changing point sizes on the phototypesetter is a slow mechanical operation. Consider size changes carefully.

14.6.2.1. Fonts and Sizes Reference

.sz +*P* The point size is set to *P* [10p], and the line spacing is set proportionally. The ratio of line spacing to point size is stored in **\n($r**. The ratio used internally by displays and annotations is stored in **\n($R**, although '.sz' does not use this.

.r *W X* Set *W* in roman font, appending *X* in the previous font. To append different font requests, use '*X* = \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 is a true underlining, as opposed to the '.ul' request, which changes to 'underline font' (usually italics in ***troff***). It won't work right if *W* is spread or broken, which includes being hyphenated, so in other words, it is only safe in nofill mode.

.q *W X* Quote *W* and append *X*. In ***nroff*** this just surrounds *W* with double quote marks (' "" '), but in ***troff*** uses directed 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, which includes being hyphenated, so in other words, it is only safe in nofill mode.

.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, which includes being hyphenated, so in other words, it is only safe in nofill mode.

*14.6.3. Quotes — *(lq' and *(rq'*

It looks better to use pairs of grave and acute accents to generate double quotes, rather than the double quote character ('"') on a phototypesetter. For example, compare "quote" to "quote". In order to make quotes compatible between the typesetter and the workstation or a terminal, 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"
```

which generates "quoted text". Notice that you must surround the material to be quoted with double quote marks if it is more than one word.

14.7. Adjusting Macro Parameters

You may adjust a number of macro parameters. You may set fonts 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, such as:

```
\f8
```

All distances are 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.

You may use registers and strings of the form '$ *x*' in expressions but you should not change them. Macros of the form '$ *x*' perform some function as described and may be redefined to change this function. This may be a sensitive operation; look at the body of the original macro before changing it.

On daisy wheel type printers in twelve pitch, you can use the '–rx1' flag to make lines default to one eighth inch, which is 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.

14.8. Roff Support

.ix +*N* Indent, no break. Equivalent to '' in *N*'.

.bl *N* Leave *N* contiguous white spaces, 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'.

.ar Set page number in arabic. Equivalent to '.af % 1'.

.n1 Number lines in margin from one on each page.

.n2 *N* Number lines from *N*, stop if N = 0.

.sk Leave the next output page blank, except for headers and footers. Use this to leave space for a full-page diagram which is produced externally and 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 generated immediately if there is room, and will otherwise be generated at the top of the next page. However, be warned: if *N* is greater than the amount of available space on an empty page, no space will ever be generated.

14.9. Preprocessor Support

.EQ *m T* Begin equation. The equation is centered if *m* is 'C' or omitted, indented **\n(bi** [4n] if *m* is 'I', and left justified if *m* is 'L'. *T* is a title printed on the right margin next to the equation. See the ***Typesetting Mathematics with 'eqn'*** chapter in this manual.

.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 **\n(es** [0.5v in ***troff***, 1v 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 large table which will not fit on one page, use h = H and follow the header part to be printed on every page of the table with a '.TH'. See the ***Formatting Tables with 'tbl'*** chapter in this manual.

.TH With '.TS H', ends the header portion of the table.

.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 '.(z' and '.)z'.

14.10. Predefined Strings

****** Footnote number, actually ***[\n($f*]**. This macro is incremented after each call to '.)f'.

***#** Delayed text number. Actually **[\n($d]**.

***[** Superscript. This string gives upward movement and a change to a smaller point size if possible, otherwise it gives the left bracket character (' ['). Extra space is left above the line to allow room for the superscript. For example, to produce a superscript you can type **x*[2*]**, which will produce x^2.

***]** Unsuperscript. Inverse of ***[**.

***<** Subscript. Defaults to '<' if half-carriage motion not possible. Extra space is left below the line to allow for the subscript.

***>** Inverse to ***<**.

***(dw** The day of the week, as a word.

***(mo** The month, as a word.

***(td** Today's date, directly printable. The date is of the form September 16, 1983. Other forms of the date can be used by using **\n(dy** (the day of the month; for example, 16), ***(mo** (as noted above) or **\n(mo** (the same, but as an ordinal number; for example, September is 9), and **\n(yr** (the last two digits of the current year).

***(lq** Left quote marks; double quote in *nroff*.

***(rq** Right quote marks; double quote in *nroff*.

***—** An em dash in *troff*; two hyphens in *nroff*.

14.11. Miscellaneous Requests

.re Reset tabs. Set to every 0.5i in *troff* and every 0.8i in *nroff*.

.ba +*N* Set the base indent to +N [0] (saved in **\n($i**). All paragraphs, sections, and displays come out indented by this amount. Titles and footnotes are unaffected. The '.H' request performs a '.ba' request if **\n(si** [0] is not zero, and sets the base indent to **\n(si*\n($0**.

.xl +*N* Set the line length to *N* [6.0i]. This differs from '.ll' because it only affects the current environment.

.ll +*N* Set line length in all environments to *N* [6.0i]. Do not use this after output has begun, and particularly not in two-columned 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 another set of macros in */usr/lib/me/local.me* which is a set of locally defined macros. These macros should all be of the form '.* *X*', where *X* is any letter (upper or lower case) or digit.

14.12. Special Characters and Diacritical Marks — '.sc'

There are a number of special characters and diacritical marks, such as accents, available with *—me*. To use these characters, you must call the macro '.sc' to define the characters before using them.

.sc Define special characters and diacritical marks. You must state this macro before initialization.

The special characters available are listed below.

Table 14-1: Special Characters and Diacritical Marks

Name	Usage	Example	
Acute accent	*´	a*´	´a
Grave accent	*`	e*`	`e
Umlaut	*:	u*:	:u
Tilde	*~	n*~	~n
Caret	*^	e*^	^e
Cedilla	*,	c*,	,c
Czech	*v	e*v	e
Circle	*o	A*o	Å
There exists	*(qe		
For all	*(qa		

14.13. '—me' Request Summary

Table 14-2: **—me** Request Summary

Request	Initial Value	Cause Break	Explanation
Request	**Cause** Break	**If no** Argument	**Explanation**
.(c	–	yes	Begin centered block.
.(d	–	no	Begin delayed text.
.(f	–	no	Begin footnote.
.(l	–	yes	Begin list.
.(q	–	yes	Begin major quote.
.(x *x*	–	no	Begin indexed item in index *x*.
.(z	–	no	Begin floating keep.
.)c	–	yes	End centered block.
.)d	–	yes	End delayed text.
.)f	–	yes	End footnote.
.)l	–	yes	End list.
.)q	–	yes	End major quote.
.)x	–	yes	End index item.
.)z	–	yes	End floating keep.
.++ *m H*	–	no	Define paper section. *m* defines the part of the paper and can be C (chapter), A (appendix), P (preliminary, for example, abstract, table of contents, etc.), B (bibliography), RC (chapters renumbered from page one each chapter), or RA (appendix renumbered from page one).
.+c *T*	–	yes	Begin chapter (or appendix, etc., as set by '.++'). *T* is the chapter title.
.1c	1	yes	One column format on a new page.
.2c	1	yes	Two column format.
.EN	–	yes	Space after equation produced by ***eqn*** or ***neqn***.
.EQ *x y*	–	yes	Precede equation; break out and add space. Equation number is *y*. The optional argument *x* may be *I* to indent equation (default), L to left-adjust the equation, or C to center the equation.
.TE	–	yes	End table.
.TH	–	yes	End heading section of table.
.TS *x*	–	yes	Begin table; if *x* is H, table has repeated heading.

.ac *A N*	–	no	Set up for ACM style output. *A* is the Author's name(s), *N* is the total number of pages. Must be given before the first initialization.
.b *x*	no	yes	Print *x* in boldface; if no argument switch to boldface.
.ba +*n*	0	yes	Augments the base indent by *n*. This indent is used to set the indent on regular text (like paragraphs).
.bc	no	yes	Begin new column.
.bi *x*	no	no	Print *x* in bold italics (nofill only).
.bx *x*	no	no	Print *x* in a box (nofill only).
.ef '*x*'*y*'*z*'	''''	no	Set even footer to x y z.
.eh '*x*'*y*'*z*'	''''	no	Set even header to x y z.
.fo '*x*'*y*'*z*'	''''	no	Set footer to x y z.
.he '*x*'*y*'*z*'	''''	no	Set header to x y z.
.hl	–	yes	Draw a horizontal line.
.hx	–	no	Suppress headers and footers on next page.
.i *x*	no	no	Italicize *x*; if *x* is missing, italic text follows.
.ip *x y*	no	yes	Start indented paragraph, which hanging tag *x*. Indentation is *y* ens (default 5).
.lp	yes	yes	Start left-block paragraph.
.lo	–	no	Read in a file of local macros of the form '.*x'. Must be given before initialization.
.np	1	yes	Start numbered paragraph.
.of '*x*'*y*'*z*'	''''	no	Set odd footer to x y z.
.oh '*x*'*y*'*z*'	''''	no	Set odd header to x y z.
.pd	–	yes	Print delayed text.
.pp	no	yes	Begin paragraph. First line indented.
.r	yes	no	Roman text follows.
.re	–	no	Reset tabs to default values.
.sc	–	no	Read in a file of special characters and diacritical marks. Must be given before initialization.
.sh *n x*	–	yes	Section head follows, font automatically bold. *n* is level of section, *x* is title of section.
.sk	no	no	Leave the next page blank. Only one page is remembered ahead.
.sz + *n*	10p	no	Increase the point size by *n* points.
.th	no	no	Produce the paper in thesis format. Must be given before initialization.
.tp	no	yes	Begin title page.
.u *x*	–	no	Underline argument (even in *troff*) (nofill only).
.uh	–	yes	Like '.sh' but unnumbered.

.xp *x*	–	no	Print index *x*.

Chapter 15

Formatting Documents with *nroff* and *troff*

15.1. Introduction to nroff and troff

nroff and ***troff*** are text processing utilities for the Sun system. ***nroff*** formats text for typewriter-like terminals (such as Diablo printers). ***troff*** is specifically oriented to formatting text for a phototypesetter. ***nroff*** and ***troff*** accept lines of text (to be printed on the final output device) interspersed with lines of format control information (to specify how the text is to be laid out on the page) and format the text into a printable, paginated document having a user-designed style. ***nroff*** and ***troff*** offer unusual freedom in document styling, including:

- detailed control over page layout;
- arbitrary style headers and footers;
- arbitrary style footnotes;
- multiple automatic sequence numbering for paragraphs, sections, etc;
- multiple column output;
- dynamic font and point-size control;
- arbitrary horizontal and vertical local motions at any point;
- a family of automatic overstriking, bracket construction, and line drawing functions.

nroff and ***troff*** are highly compatible with each other and it is almost always possible to prepare input acceptable to both. The formatters provide requests (conditional input) so that you can embed input expressly destined for either ***nroff*** or ***troff***. ***nroff*** can prepare output directly for a variety of terminal types and is capable of utilizing the full resolution of each terminal.

This paper[1] provides a user's guide and reference section for ***nroff*** and ***troff***. Note that throughout the text we refer to ***nroff*** and ***troff*** more-or-less interchangeably — places where the narrative refers specifically to one or the other processor are noted.

You should be aware that using ***nroff*** or ***troff*** 'in the raw' requires a detailed knowledge of the way that these programs work and a certain knowledge of typographical terms. ***nroff*** and ***troff*** don't do a great deal of work for you — for example, you have to explicitly tell them how to do paragraph indents and page numbers and things like that. If what you are trying to do is just get a job done (like writing a memo), you shouldn't be reading this chapter at all, but instead should be reading the chapter in this manual entitled ***Formatting Documents with the*** **–ms** ***Macro Package***. If, on the other hand, you would like to learn the fine details of a programming

[1] The material in this chapter evolved from *A troff Tutorial*, by Brian Kernighan of Bell Laboratories, and from *nroff/troff User's Manual*, originally written by Joseph Ossanna of Bell Laboratories.

language designed to control a typesetter, this is the place to start reading. In many ways, ***nroff***'s and ***troff***'s control language resembles an assembly language for a computer — a remarkably powerful and flexible one — but nonetheless such that many operations must be specified at a level of detail and in a form that is too hard for most people to use effectively. The single most important rule of using ***troff*** is not to use it directly, but through some intermediary such as one of the macro packages described previously, or via one of the various preprocessors. In the few cases where existing packages don't do the whole job, the solution is ***not*** to write an entirely new set of ***troff*** instructions from scratch, but to make small changes to adapt packages that already exist. In accordance with this philosophy of letting someone else do the work, the part of ***troff*** described here is only a small part of the whole, although it tries to concentrate on the more useful parts. In any case, there is no attempt to be complete. Rather, the emphasis is on showing how to do simple things, and how to make incremental changes to what already exists.

15.1.1. Text Formatting Versus Word Processing

Many newcomers to the UNIX† system are surprised to find that there are no word processors available. This is largely historical — the types of documents (such as the Sun manuals) that people do with the UNIX system's text formatting packages just can't be done with existing word processors. Before you get into the details of ***nroff*** and ***troff***, here is a short discussion on the differences between text formatters and word processors, and their relative strengths and weaknesses.

A ***word processor*** is a program that to some extent simulates a typewriter — text is edited and formatted by one program. You type text at a computer terminal, and the word processor formats the text on the screen for you as you go. You usually get special effects like underlining and boldfacing by typing control indicators, and the word processor shows this by inverting the screen or something like that. The document is displayed on the terminal screen in the same format as it will appear on the printing device. The effects of this are often termed 'What You See Is What You Get' (usually called WYSIWIG and pronounced 'wizzi-wig'). Unfortunately, as has been pointed out, the problem with many WYSIWIG editors is that 'What You See Is *All* You Get'. In general, word processors cannot handle large documents. In principle it is possible to write large manuals and even whole books with word processors, but the process gets painful for large manuscripts. Sometimes a change, such as deleting a sentence or inserting a new one, in the early part of a document can require that the whole document has to be reformatted. A change in the overall structure of the formatting requirements (for example, a changed indentation depth) will also mean that the whole document has to be reformatted. Word processors, in general, don't cope with automatic chapter and section numbering (of the kind you see in the Sun manuals), neither can they generate tables of contents and indices automatically. These tasks have to be done manually, and are a potential source of error. Word processors are eminently suitable for memos and letters, and can handle short documents. But large documents, or formatting documents for sophisticated devices like modern phototypesetters, requires a text formatter.

A ***text formatter*** such as ***nroff*** or ***troff*** does not in general do any editing functions — its only job in life is reading text from a file and formatting that text ready for printing on some printing device. Entering the text into the file, and formatting the text from that file ready for printing are two separate and independent operations. You prepare your file full of text using a text

[1] † UNIX is a trademark of Bell Laboratories.

editor such as *vi* (described elsewhere in this manual). The file contains text to be formatted, interspersed with formatting instructions which control the layout of the final text. The text formatter reads this file of text, and obeys the formatting instructions contained in the file. The results of the formatting process is a finished document. The disadvantage of a text formatter is that you have to run them to find out what the final result will look like. Many people find the idea of embedded 'formatting commands' foreign, as they do the idea of two separate processes (an edit followed by a run of the formatter) to get the final document.

Notwithstanding all of the above, the UNIX system has had text formatting utilities since the very beginning, and the UNIX system has many documents written all using the capabilities of *nroff* or *troff*.

15.1.2. The Evolution of nroff and troff

One of the very first text formatting programs was called *runoff* and was a utility for the Compatible Time Sharing System (CTSS) at MIT in the early 1960's. *Runoff* was named for the way that people would say 'I'll just run off a document'.

When the UNIX system came to have a text formatter, the text formatter was called *roff*, because UNIX people like to call things by short and cryptic names. *Roff* was a simple program that was easy to work with as long as you were writing very small and simple documents for a line-printer. In some ways, *roff* is easier to use than *nroff* or *troff* because *roff* had built in facilities such as being able to specify running headers and footers for a document with simple commands.

nroff stands for 'Newer *roff*'. *troff* is an adaptation of *nroff* to drive a phototypsetting machine. Although *troff* is supposed to mean 'typesetter *roff*', some people have formed the theory that *troff* actually stands for 'Times Romanoff' because of *troff*'s penchant for the Times Roman typeface.

nroff and *troff* are much more flexible (and much more complicated) programs — it's safe to say that they don't do a lot for you — for instance, you have to manage your own pagination, headers, and footers. The way that *nroff* and *troff* ease the burden is via facilities to define your own text formatting commands (macros), define strings, and store and manipulate numbers. Without these facilities, you would go mad (many people have — the author of this document among them). In addition, there are supporting packages for doing special effects such as mathematics and tabular layouts.

15.1.3. Preprocessors and Postprocessors

Because *troff* or *nroff* are so hard to use 'in the raw', various tools have evolved to convert from human-oriented ways of specifying things into codes that *troff* or *nroff* can understand. Tools that do translations for *troff* or *nroff* before the fact are called *preprocessors*. There are also tools that hack over the output of *nroff* for different devices or for other requirements. Tools that do conversions of *troff* or *nroff* output after the fact are called *postprocessors*.

Two of the major preprocessors available for *troff* or *nroff* are called *tbl* (for assisting with laying out tables), and *eqn* (a language for specifying mathematical constructs). In traditional typesetting, tables and equations are two forms of copy layout that cause a lot of trouble and cost a bunch of money to do. *Tables* where stuff must be laid out in columns, maybe with boxes and lines around them, seem to give typographers fits. The UNIX system supplies a preprocessor called *tbl* where you describe what a table should look like in general terms, and then *tbl* and

nroff or ***troff*** together work out how to lay out the tabular material on the page. ***Mathematics*** is known in the typesetting business as 'penalty copy' because it is hard to do in the traditional way and so you must pay a penalty for the fact that the typesetting business hasn't got its act together. The UNIX system supplies a preprocessor called ***neqn*** (for use with ***nroff***) and ***eqn*** (for use with ***troff***) with which you can describe equations in a kind of English-like language, and then ***neqn*** and ***nroff*** or ***eqn*** and ***troff*** together work out how to lay out the equations on the page.

So much for the preprocessors. There are also postprocessors to hack over the output of ***nroff*** or ***troff*** when the printing device can't handle what the text formatting program is trying to do. When you use ***nroff*** to format tables with vertical lines, or to generate multiple-column output, ***nroff*** generates what are called 'reverse paper motions'. Not all printers can feed the paper in the reverse direction, and so there is a postprocessor called ***col*** which works out how to handle the reverse paper movements with only forward motions.

15.1.4. troff, Typesetters, and Special-Purpose Formatters

Please be sure to read this bit: this section covers some aspects of ***troff*** that are generally glossed over in the traditional UNIX manuals. ***troff*** was originally designed as a text formatter targeted to one specific machine — that machine was called a Graphics Systems Incorporated (GSI) C/A/T (Computer Assisted Typesetter). The C/A/T is a strange and wonderful device with strips of film mounted on a revolving drum, lenses, and light pipes. The C/A/T flashes character images on film which you then develop to produce page proofs for your book or manual or whatever. The C/A/T is almost extinct now except for some odd niches like Berkeley.

troff was written very much with the C/A/T in mind. The internal units of measurement that ***troff*** uses are C/A/T units, ***troff*** only understands four fonts at a time, and so on. Throughout this chapter, much of the terminology is based on ***troff***'s intimate relationship with the C/A/T.

15.1.5. Using the nroff and troff Text Formatters

To use ***nroff*** or ***troff*** you first prepare your file of text with ***nroff*** or ***troff*** requests embedded in the file to control the formatting actions. The remainder of this document contains a discussion on the formatting commands. Then you run the formatter at the UNIX command level like this:

```
hostname% nroff  options files
```

or, of course:

```
hostname% troff  options files
```

where ***options*** represents any of a number of option arguments and ***files*** represents the list of files containing the document 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.

Options may appear in any order so long as they appear before the files. There are three parts to the list of options below: the first list of options are common to both ***nroff*** and ***troff***; the second list of options are only applicable to ***nroff***; the third list of options are only applicable to ***troff***.

Each option is typed as a separate argument — for example,

```
hostname% nroff -o4,8-10 -T300S -msun file1 file2
```

formats 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 **sun** macro package.

15.1.5.1. Options Common to nroff and troff

-o*list*
Print only pages whose page numbers appear in *list*, which consists of comma-separated numbers and number ranges. A number range has the form *N–M* and means pages *N* through *M;* a initial *–N* means from the beginning to page *N;* and a final *N–* means from *N* to the end.

-n*N*
Number first generated page *N*.

-s*N*
Stop every *N* pages. *nroff* will halt prior to every *N* pages (default *N*=1) to allow paper loading or changing, and will resume upon receipt of a newline.

-m*name*
Prepends the macro file **/usr/lib/tmac/tmac.***name* to the input *files*.

-r*aN*
Register *a* (one-character) is set to *N*.

-i Read standard input after the input files are exhausted.

-q Invoke the simultaneous input-output mode of the **rd** request.

-z Suppress formatted output. The only output you get are messages from **.tm** (terminal message) requests, and from diagnostics.

15.1.5.2. Options Applicable Only to nroff

-h Output tabs used during horizontal spacing to speed output as well as reduce byte count. Device tab settings assumed to be every 8 nominal character widths. Default settings of input (logical) tabs is also initialized to every 8 nominal character widths.

-T*name*
Specifies the name of the output terminal type. Currently defined names are **37** for the (default) Model 37 Teletype®, **tn300** for the GE TermiNet 300 (or any terminal without half-line capabilities), **300S** for the DASI-300S, **300** for the DASI-300, and **450** for the DASI-450 (Diablo Hyterm).

-e Produce equally-spaced words in adjusted lines, using full terminal resolution.

15.1.5.3. Options Applicable Only to troff

−t Direct output to the standard output instead of the phototypesetter.

−a Send a printable (ASCII) approximation of the results to the standard output.

−p*N*
Print all characters in point size *N* while retaining all prescribed spacings and motions, to reduce phototypesetter elasped time.

15.1.6. General Explanation of troff and nroff Source Files

This section of the ***nroff/troff*** manual covers generic topics related to the format of the input file, how requests are formed, and how numeric parameters to requests are stated.

To use ***troff*** you have to prepare not only the actual text you want printed, but some information that tells ***how*** you want it printed. For ***troff*** the text and the formatting information are often intertwined. Most commands to ***troff*** are placed on a line separate from the text itself, beginning with a period (one command per line). For example:

```
Here is some text in the regular size characters, but we want
to make some of the text in a
.ps 14
larger size to emphasize something
```

changes the 'point size', that is, the size of the letters being printed, to '14 point' (one point is 1/72 inch) like this:

Here is some text in the regular size characters, but we want to make some of the text in a **larger size to emphasize something**

Occasionally, though, something special occurs in the middle of a line — to produce Area$=\pi r^2$ you have to type

```
Area = \(*p\fIr\fR\|\s8\u2\d\s0
```

(which we will explain shortly). The backslash character \ introduces ***troff*** commands and special characters within a line of text.

To state the above more formally, an input file to be processed by ***troff*** or ***nroff*** consists of ***text lines***, which are destined to be printed, interspersed with ***control lines***, which set parameters or otherwise control subsequent processing. A control line is usually called a ***request***.

A request begins with a ***control character*** — normally **.** (period) or **′** (apostrophe or acute accent) — followed by a one or two character name. A request is either:

a basic request
(also called a command) which is one of the many predefined things that ***nroff*** or ***troff*** can do. For example, **.ll 6.5i** is a basic request to set the line-length to 6.5 inches, and **.in 5** is a basic request to indent the left margin by 5 en-spaces.

a macro reference
specifies substitution of a user-defined ***macro*** in place of the request. A ***macro*** is a predefined collection of basic requests and (possibly) other macros. Macros are defined in

terms of basic requests. For example, in the **−ms** macro package discussed elsewhere in this manual, **.LP** is a macro to start a new left-blocked paragraph.

The ´ (apostrophe or acute accent) control character suppresses the ***break*** function— the forced output of a partially filled line— caused by certain requests.

The control character may be separated from the request or macro name by white space (spaces and/or tabs) for esthetic reasons. Names must be followed by either space or newline. ***nroff*** or ***troff*** *IGNORES* control lines whose names are unrecognized.

Various special functions may be introduced anywhere in the input by means of an ***escape*** character, normally \. For example, the function **\n***R* interpolates the contents of the ***number register*** whose name is *R* in place of the function. Here *R* is either a single character name in which case the escape sequence has the form **\n***x*, or else *R* is a two-character name, in which case the escape sequence must have the form **\n(***xx*. In general, there are many escape sequences whose one-character form is **\f***x* and whose two-character form is **\f(***xx*, where **f** is the function and *x* or *xx* is the name.

15.1.6.1. Backspacing

Unless in ***copy mode***, the ASCII backspace character is replaced by a backward horizontal motion having the width of the space character. Underlining as a form of line-drawing is discussed in the section on ***Arbitrary Motions and Drawing Lines and Characters***. A generalized overstriking function is also described in the above- mentioned section.

15.1.6.2. Comments

Comments may be placed at the ***end*** of any line by prefacing them with \". A comment line cannot be continued by placing a \ at the end of the line — see the discussion on continuation lines below.

A line beginning with \" appears as a blank line and behaves like a **.sp 1** request:

```
 Here is a line of text
\" and here is a comment on a line by itself
 here is another line of text
```

when we format the above lines we get this:

```
Here is a line of text

and here is another line of text
```

If you want a comment on a line by itself but you don't want it to appear as a blank line, type it as .\":

```
Here is a line of text
.\"  and here is a comment on a line by itself
and here is another line of text
```

when we format the above lines we get this:

```
Here is a line of text
and here is another line of text
```

15.1.6.3. Continuation Lines

An uncomfortably long input line that must stay one line (for example, a string definition, or unfilled text) can be split into many physical lines by ending all but the last one with the escape \. The sequence \(newline) is ***always*** ignored — except in a comment — see below. This provides a continuation line facility. The \ at the end of the line is called a ***concealed newline*** in the jargon.

15.1.6.4. Transparent Throughput

An input line beginning with a **\!** is read in ***copy mode*** and ***transparently*** output (without the initial **\!**); the text processor is otherwise unaware of the line's presence. This mechanism may be used to pass control information to a post-processor or to embed control lines in a macro created by a diversion.

15.1.6.5. Formatter and Device Resolution

troff internally uses 432 units/inch, corresponding to the phototypesetter which has a horizontal resolution of 1/432 inch and a vertical resolution of 1/144 inch. ***nroff*** internally uses 240 units/inch, corresponding to the least common multiple of the horizontal and vertical resolutions of various typewriter-like output devices. ***troff*** rounds horizontal/vertical numerical parameter input to the actual horizontal/vertical resolution of the Graphic Systems typesetter. ***nroff*** similarly rounds numerical input to the actual resolution of the output device indicated by the **−T** option (default Model 37 Teletype).

15.1.6.6. Specifying Numerical Parameters

Many requests can have numerical arguments. Both ***nroff*** and ***troff*** accept numerical input in a variety of units. The general form of such input is

```
.xx nnnnunits
```

where **.xx** is the request, ***nnnn*** is the number, and **units** is the ***scale indicator***.

Scale indicators are 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.

Table 15-1: Scale Indicators for Numerical Input

Scale Indicator	*Meaning*	*Number of basic units* *troff*	*nroff*
i	Inch	432	240
c	Centimeter	432X50/127	240X50/127
P	Pica = 1/6 inch	72	240/6
m	Em = *S* points	6X*S*	*C*
n	En = Em/2	3X*S*	*C, same as Em*
p	Point = 1/72 inch	6	240/72
u	Basic unit	1	1
v	Vertical line space	*V*	*V*
none	Default, see below		

In ***nroff***, ***both*** the em and the en are taken to be equal to the *C*, which is output-device dependent; 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 already 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, *N*, 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 ¦ 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*** (see the section on diversions) to the vertical place *N*. For ***all*** other requests and functions, ¦*N* becomes the distance from the current horizontal place on the ***input*** line to the horizontal place *N*. For example,

```
.sp  |3.2c
```

will space ***in the required direction*** to 3.2 centimeters from the top of the page.

15.1.6.7. Numerical Expressions

Wherever numerical input is expected, you can type an arithmetic expression. An expression involves parentheses and the arithmetic operators and logical operators shown in the table below

Table 15-2: Arithmetic Operators and Logical Operators for Expressions

Arithmetic Operator	*Meaning*
+	Addition
−	Subtraction
/	Division
*	Multiplication
%	Modulo
Logical Operator	***Meaning***
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
= **or** ==	Equal to
&	and
:	or

Except where controlled by parentheses, evaluation of expressions is left-to-right — there is no operator precedence.

In 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 1/2 the sum of 4.25 inches + 2 picas + 30 points.

15.1.7. Notation Used in this Manual

Numerical parameters are indicated in this manual 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 lower case letters and one/two character arguments are indicated by a pair of lower case letters. Character string arguments are indicated by multi-character mnemonics.

15.1.8. Output and Error Messages

The output from **tm**, **pm**, and the prompt from **rd**, as well as various ***error*** messages are written onto UNIX's ***standard error message*** output. The latter is different from the ***standard output***, where ***nroff*** formatted output goes. By default, both are written onto the user's terminal, but they can be independently redirected — in the case of ***troff***, the standard output should always be redirected unless the **−a** option is in effect, because ***troff***'s output is a strange binary format destined to drive a typesetter.

Various ***error*** conditions may occur during the operation of ***nroff*** and ***troff***. Certain less serious errors having only local impact do not stop processing. 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 philosophy is to continue processing, if possible, on the grounds that output useful for debugging may be produced. 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.

15.2. Filling and Adjusting Lines of Text

Perhaps the most important reason for using ***troff*** or ***nroff*** is to use its filling and adjusting capabilities. Here is a summary of what filling and adjusting mean:

Filling means that ***troff*** or ***nroff*** collects ***words*** from your ***input*** text lines and assembles the collected words 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. Filling continues until something happens to break the filling process, such as a blank line in the text, or one of the ***troff*** or ***nroff*** requests that break the line — things that break the filling process are discussed later on.

Adjusting means that once the line has been filled as full as possible, spaces between words on the output line are then increased to spread out the line to the current ***line-length*** minus any current ***indent***. The paragraphs you have just been reading are both filled and adjusted. Justification implies filling — it makes no sense to adjust lines without also filling them.

In the absence of any other information, ***troff***'s or ***nroff***'s standard behavior is to fill lines and adjust for straight left and right margins, so it is quite possible to create a neatly formatted document which only contains lines of text and no formatting requests. Given this as a starting point, the simplest document of all contains nothing but blocks of text separated by blank lines — ***troff*** or ***nroff*** will fill and justify those blocks of text into paragraphs for you. To get further control over the layout of text, you have to use ***requests*** and ***functions*** embedded in the text, and that is the subject of this entire paper on using ***troff***.

A ***word*** is any string of characters delimited by the ***space*** character or the beginning or 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 ***unpaddable space*** character '\ ' (backslash-space) — also called a 'hard blank' in other systems. The adjusted word spacings are uniform in ***troff*** and the minimum interword spacing can be controlled with the **.ss** (**s**pace **s**ize) request. In ***nroff***, interword spaces are normally nonuniform because of quantization to character-size spaces, but the **−e** command line option requests uniform spacing to the full resolution of the output device. Multiple inter-word space characters found in the input are retained, except for trailing spaces.

Filling and adjusting and hyphenation can all be prevented or controlled by requests that are discussed later in this part of the manual.

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.

A text input line that happens to begin with a control character can be made to not look like a control line by prefacing it with the non-printing, zero-width filler character **\&**. Still another way is to specify output translation of some convenient character into the control character using the **.tr** (translate) request — see the relevant section.

The ***text length*** on the last line output is available in the **.n** number register, and text base-line position on the page for this line is in the **nl** number register. The text base-line high-water mark (lowest place) on the current page is in the **.h** number register.

15.2.1. Controlling Line Breaks

When filling is turned on, words of text are taken from input lines and placed on output lines to make the output lines as long as they can be without overflowing the line length, until something happens to break the filling process. When a break occurs, the current output line is printed just as it is, and a new output line is started for the following input text. There are various things that cause a break to occur:

Table 15-3: Constructs that Break the Filling Process

Construct	*Explanation*
Blank line(s)	If your input text contains any completely blank lines, ***troff*** or ***nroff*** assumes you mean them. So it prints the current output line, then your blank lines, then starts the following text on a new line.
Spaces	at the beginning of a line are significant. If there are spaces at the start of a line, ***troff*** or ***nroff*** assumes you know what you are doing and that you really want spaces there. Obviously, to achieve this, the current output line must be printed and a new line begun. Avoid using tabs for this purpose, since they do not cause a break.
A **.br** request	A **.br** request (**br**eak) request can be used to make sure that the following text is started on a new line.
troff or nroff requests	Some ***troff*** or ***nroff*** requests cause a break in the filling process. However, there is an alternate format of these requests which does not cause a break. That is the format where the initial period character (.) in the request is replaced by the apostrophe or single quote character ('). The list of requests that cause a break appears in the table below this one.
A **\p** *Function*	When filling is in effect, the in-line **\p** function may be embedded 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.
End of file	Filling stops when the end of the input file is reached.

Breaks caused by blank lines or spaces at the beginning of a line enable you to take advantage of the filling and justification features provided by ***troff*** or ***nroff*** without having to use any ***troff*** or ***nroff*** requests in your text.

As mentioned in the table above in the item entitled ***troff or nroff requests***, there are some requests that cause a break when they are encountered. The list of requests that break lines is short and natural:

Table 15-4: Formatter Requests that Cause a Line Break

Command	*Explanation*
.bp	Begin a new page
.br	Break the current output line
.ce	Center line(s)
.fi	Start filling text lines
.nf	Stop filling text lines
.sp	Space vertically
.in	Indent the left margin
.ti	Temporary indent the left margin for the next line only

No other requests break lines, regardless of whether you use a **.** or a **'** as the control character. If you really *do* need a break, add a **.br** (**br**eak) request at the appropriate place, as described below.

15.2.1.1. **.br** — *Break Lines*

The **.br** (**br**eak) request breaks the current output line and stops filling that line. Any new output will start on a new line.

Summary of the .br Request

Item	*Description*
Form of Request:	**.br**
Initial Value:	Not Applicable
If No Argument:	This request doesn't require an argument.
Explanation:	Stop filling the line currently being collected and output the line without adjustment. Text lines beginning with space characters and empty text lines (blank lines) also cause a break.

15.2.2. Continuation Lines and Interrupted Text

The copying of an input line in ***nofill*** (non-fill) mode (see below) can be ***interrupted*** by terminating the partial line with a **\c**. The ***next*** encountered input text line 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.

The next section talks about the different ways of getting text adjusted.

15.2.3. **.ad** — *Specify Adjusting Styles*

To change the style of adjusting text, you use the **.ad** (**ad**just) request to specify one of the four different methods for adjusting text:

Table 15-5: Adjusting Styles for Filled Text

Adjusting Indicator	*Adjusting Style*	*Description*
.ad l	Left	Produces flush-left, ragged-right output, which is the same as filling with no adjustment.
.ad r	Right	Produces flush-right, ragged-left output.
.ad c	Center	Centers each output line, giving both left and right ragged margins.
.ad b	Both	Justifies both left and right margins.
.ad n	Normal	
.ad	Reset	Resumes adjusting lines in the last mode requested.

It makes no sense to try to adjust lines when they are not being filled, so if filling is off when a **.ad** request is seen, the adjusting is deferred until filling is turned on again.

Summary of the .ad Request

Item	*Description*
Form of Request:	**.ad** *c*
Initial Value:	**.ad b** — that is, adjust both margins.
If No Argument:	Adjust in the last specified adjusting mode.
Explanation:	Adjust lines — if fill mode is off, adjustment is be deferred until fill mode is back on. If the type indicator *c* is present, the adjustment type is changed as shown in the following table.
Notes:	E

The current adjustment indicator *c* can be obtained from the **.j** number register.

The figure below illustrates the different appearances of filled and justified text.

This paragraph is *filled* and *adjusted* on both margins. This is the easiest formatting style to achieve using *nroff* or *troff* because you don't have to place any requests in your text — you just type the blocks of text into the input file and the formatter does something reasonably sane with them. Although we specified nothing to get the paragraph filled and adjusted, we could have used an **.ad b** (**ad**just **b**oth) request, or an **.ad n** (**ad**just **n**ormal) request — they both mean the same thing, namely, fill lines and adjust both margins.

This paragraph is an example of 'flush left, ragged right', which is what you get when you have filling without adjusting — words are placed on the line to fill lines out as far as possible, but no interword spaces are inserted so the right-hand margin looks ragged. This paragraph was formatted using an **.ad l** (**ad**just **l**eft) request, which has the same effect as using a **.na** (**n**o **a**djust) request described later.

Then this paragraph is an illustration of text formatted as 'flush right, ragged left' — words are placed on the line to fill lines out as far as possible, then the lines are made to line up on the right-hand margin, no interword spaces are inserted, and so the left-hand margin looks ragged. This paragraph was formatted using an **.ad r** (**ad**just **r**ight) request.

Finally, this paragraph is an instance of a formatting style called 'centered' adjusting, also known as 'ragged left, ragged right' — words are placed on the line to fill lines out as far as possible, then the lines are *centered* so that both margins look ragged. This paragraph was formatted using an **.ad c** (**ad**just **c**enter) request.

Figure 15-1: Filling and Adjusting Styles

15.2.4. **.na** — *No Adjusting*

If you don't specify otherwise, *troff* or *nroff* justifies your text so that both left and right margins are straight. This can be changed if necessary — one way, as we showed above, is to use the **.ad l** request to get left adjusting only so that the left margin is straight and the right margin is ragged. Another way to achieve this same effect is to use the **.na** (**n**o **a**djust) request. Output lines are still filled, providing that filling hasn't also been turned off — see the **.nf** (**n**o **f**ill) request below. If filling is still on, *troff* or *nroff* produces flush left, ragged right output.

Summary of the .na Request	
Item	*Description*
Form of Request:	**.na**
Initial Value:	Adjusting is on by default
If No Argument:	No argument required — adjustiung is turned off
Explanation:	Turn off adjustment — the right margin will be ragged. The adjustment type for the **.ad** request is not changed. Output lines are still *filled* if fill mode is on.
Notes:	E

15.2.5. **.nf** *and* **.fi** *— Turn Filling Off and On*

The **.nf** (**n**o **f**ill) request turns off filling. Lines in the result are neither filled nor adjusted. The output text appears exactly as it was typed in, complete with any extra spaces and blank lines you might type — this is often called 'as-is text', or 'verbatim'. No filling is mainly used for showing examples, especially in computer books where you want to show examples of program source code.

You should be aware that traditional typesetting people have trouble with the concept of no filling, because their typesetting systems are geared up to fill and adjust text all the time. When you ask for stuff to be printed exactly the way you typed it, they have problems, especially when you want blank lines left in the unfilled text exactly where you put them. In the world of typography, things that don't fit into the Procrustean mold of filled text are often called 'displays' and have to be handled specially.

The **.fi** (**fi**ll) request turns on **fi**lling. If adjusting has not been turned off by a **.na** request, output lines are also adjusted in the prevailing mode set by any previous **.ad** request.

Summary of the .fi Request	
Item	*Description*
Form of Request:	**.fi**
Initial Value:	Filling is on by default
If No Argument:	No argument required — filling is turned on
Explanation:	Fill subsequent output lines. The register **.u** is 1 in fill mode and 0 in nofill mode.
Notes:	E,B

Summary of the .nf Request

Item	*Description*
Form of Request:	**.nf**
Initial Value:	Filling is on by default
If No Argument:	No argument required — filling is turned off
Explanation:	Subsequent output lines are ***neither*** filled ***nor*** adjusted. Input text lines are copied directly to output lines ***without regard*** for the current line length. The register **.u** is 1 in fill mode and 0 in nofill mode.
Notes:	E,B

15.2.6. Hyphenation

When *troff* or *nroff* fills lines, it takes each word in turn from the input text line, and puts the word on the output text line, until it finds a word which will not fit on the output line. At this point *troff* or *nroff* tries to hyphenate the word. If possible, the first part of the hyphenated word is put on the output line followed by a –, and the remainder of the word is put on the next line. At this point we should emphasize that, although we have been showing the examples both filled and justified, it is during *filling* that *troff* or *nroff* hyphenates words, not adjusting.

If you have in your input text words containing a hyphen (such as jack-in-the-box, or co-worker), *troff* or *nroff* will if necessary split these words over two lines, regardless of whether hyphenation is turned off.

15.2.6.1. **.nh** *and* **.hy** *— Control Hyphenation*

Normally, when you invoke *troff* or *nroff*, hyphenation is turned on, but you can change this. The **.nh** (**n**o **h**yphenation) request turns off automatic hyphenation. The only words that are split over more than one line are those which already contain '–'. Hyphenation can be turned on again with the **.hy** (**h**yphenate) request.

.hy can be given an argument to restrict the amount of hyphenation that *troff* or *nroff* does. The argument is numeric. The request **.hy 2** stops *troff* or *nroff* from hyphenating the last word on a page. **.hy 4** instructs *troff* or *nroff* not to split the last two characters from a word; so, for example, 'repeated' will never be hyphenated 'repeat-ed'. **.hy 8** requests the same thing for the first two characters of a word; so, for example, 'repeated' will not be hyphenated 're-peated'.

The values of the arguments are additive: **.hy 12** makes sure that words like 'repeated' will never be hyphenated either as 'repeat-ed' or as 're-peated'. **.hy 14** calls up all three restrictions on hyphenation.

A **.hy 1** request is the same as the simple **.hy** request — it turns on hyphenation everywhere. Finally, a **.hy 0** request is the same as the **.nh** request — it turns off automatic hyphenation altogether.

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), 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 or off.

Summary of the .nh Request

Item	*Description*
Form of Request:	**.nh**
Initial Value:	Hyphenation is on by default
If No Argument:	No argument required — hyphenation is turned off
Explanation:	Turn automatic hyphenation off.
Notes:	E

Summary of the .hy Request

Item	*Description*
Form of Request:	**.hy** *N*
Initial Value:	Hyphenation is on by default
If No Argument:	$N=1$.
Explanation:	Turn automatic hyphenation on for $N \geq 1$, or off for $N=0$. If $N=2$, do not hyphenate ***last*** lines (ones that cause a trap). If $N=4$, do not hyphenate the ***last*** two characters of a word. If $N=8$, do not hyphenate the ***first*** two characters of a word. These values are additive — that is, $N=14$ invokes all three restrictions.
Notes:	E

15.2.6.2. **.hw** — *Specify Hyphenation Word List*

If there are words that you want ***troff*** or ***nroff*** to hyphenate in some special way, you can specify them with the **.hw** (**h**yphenate **w**ords) request. This request tells ***troff*** or ***nroff*** that you have special cases it should know about, for example:

```
.hw pre-empt ant-eater
```

Now, if either of the words 'preempt' or 'anteater' need to be hyphenated, they will appear as specified on the **.hw** request, regardless of what ***troff*** or ***nroff***'s usual hyphenation rules would do. If you use the **.hw** request, be aware that there is a limit of about 128 characters in total, for the list of special words.

Summary of the .hw Request

Item	*Description*
Form of Request:	**.hw** *word1 ...*
Initial Value:	None
If No Argument:	Ignored
Explanation:	Specify hyphenation points in words with embedded minus signs. Versions of a word with terminal *s* are implied — that is, ***dig—it*** implies ***dig—its***. This list is examined initially ***and*** after each suffix stripping. The space available is small — about 128 characters.

15.2.6.3. **.hc** — *Specify Hyphenation Character*

A ***hyphenation indicator*** character may be embedded in a word to specify desired hyphenation points, or may be prepended to suppress hyphenation.

Summary of the .hc Request

Item	*Description*
Form of Request:	**.hc** *c*
Initial Value:	\%
If No Argument:	\%
Explanation:	Set hyphenation indicator character to *c* or to the default \%. The indicator does not appear in the output.
Notes:	E

15.2.7. **.ce** — *Center Lines of Text*

When we described ***Filling and Adjusting***, we showed how the text produced by ***nroff*** or ***troff*** could be centered by using the **.ad c** request. Setting text adjustment for centering is a fairly unusual way of getting centered text, because the text is being filled at the same time. The more usual use for centering is to have unfilled lines that are centered — that is, each line that you type is centered within the output line. You get lines centered via the **.ce** (**ce**nter) request, which **ce**nters lines of text.

If you just use a **.ce** request without an argument, ***troff*** or ***nroff*** centers the ***next*** line of text:

```
.ce
```

centers the following line of text, whereas:

```
.ce 5
```

centers the following five lines of text. Filling is temporarily turned off when lines are centered, so each line in the input appears as a line in the output, centered between the left and right margins. For centering purposes, the left margin includes both the page offset (see later) and any indentation (also see later) that may be in effect.

An argument of zero to the **.ce** request simply stops any centering that might be in progress. So, if you don't want to count how many lines you want centered, you can ask for some large number of lines to be centered, then follow the last of the lines with a **.ce 0** request:

```
.ce 100
.
.
.
lines of text to be centered
.
.
.
.ce 0
```

The '100' in the example above could be any large number that you think is bigger than the number of lines to center.

Note that the argument to the **.ce** request only applies to following ***text*** lines in the input. Lines containing ***nroff*** or ***troff*** requests are not counted.

Summary of the .ce Request

Item	*Description*
Form of Request:	**.ce** *N*
Initial Value:	Centering is off by default.
If No Argument:	*N*=1
Explanation:	Center the next *N* input text lines within the current line (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 is left adjusted.
Notes:	E,B

15.2.8. **.ul** *and* **.cu** — *Underline or Emphasize Text*

There are times when you want to lend ***emphasis*** to a word in a piece of text. The normal way to do this is to place the word or piece of text in ***italics*** if you have an italic font, or underline the word if you don't have an italic font. The **.ul** (**u**nderline) request underlines alphanumeric characters in ***nroff***, and prints those characters in the italic font in ***troff***. As with the **.ce** request, a **.ul** request with no argument underlines a single line of text, so:

```
.ul
```

simply underlines the following line of text. A numeric argument to the **.ul** request specifies the number of text lines you want underlined, so:

.ul 3

underlines the next three lines of text. As with centering, an argument of zero **.ul 0** cancels the underlining process.

Another form of underlining is called up with the **.cu** request, and asks for **c**ontinuous **u**nderlining. This is the same as the **.ul** request, except that ***all*** characters are underlined. Again, if you are using ***troff*** the characters are printed in the italic font instead of underlined. There is a way to get characters underlined in ***troff***, and this technique is explained later in this manual.

As with **.ce**, only lines of text to be underlined are counted in the number given to the underline request. ***nroff*** or ***troff*** requests interspersed with the text lines are not counted.

15.2.9. Underlining

nroff automatically underlines characters in the ***underline*** font, specifiable with a **.uf** (**u**nderline **f**ont) request. The underline font is normally Times Italic and is mounted on font position 2 In addition to the **.ft** (**f**o**nt**) request and the **\f***F*, the underline font may be selected by the **.ul** (**u**nderline) request and the **.cu** (**c**ontinuous **u**nderline) request. Underlining is restricted to an output-device-dependent subset of ***reasonable*** characters.

Summary of the .ul Request

Item	*Description*
Form of Request:	**.ul** *N*
Initial Value:	Underlining is off by default.
If No Argument:	*N*=1
Explanation:	Underline in ***nroff*** (italicize in ***troff***) the next *N* input text lines. Actually, switch to ***underline*** 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 a **.tl** request ***is*** affected by the font change, but does ***not*** decrement *N*. If *N*>1, there is the risk that a trap interpolated macro may provide text lines within the span — environment switching can prevent this.
Notes:	E

Summary of the .cu Request

Item	*Description*
Form of Request:	**.cu** *N*
Initial Value:	Underlining is off by default.
If No Argument:	*N*=1
Explanation:	A variant of **ul** that underlines *every* character in *nroff*. Identical to **ul** in *troff*.
Notes:	E

Summary of the .uf Request

Item	*Description*
Form of Request:	**.uf** *F*
Initial Value:	Italic
If No Argument:	Italic
Explanation:	Set underline font to *F*. In *nroff*, *F* may *not* be on position 1 (initially Times Roman).

15.3. Controlling Page Layout

Now we get into the subject of altering the physical dimensions of the layout of text on a page. There are two major parts to page control, and they can be roughly divided into controlling the ***horizontal*** aspects of lines, and controlling the ***vertical*** aspects of the page dimensions.

Horizontal page control
: Deals with subjects such as the location of the left margin, the location of the right margin (the length of the line), and indentation of lines.

Vertical page control
: Deals with the physical length of the page, when pages get started, and whether there's enough room on the current page for a block of text. Page numbering is also covered in this area.

These topics are covered in this section. We deal first with horizontal page control, then with the vertical aspects of page control.

We should explain how ***troff*** thinks of a page. The next page contains a diagram of a page of text, and here we explain what some of the terms mean:

Page Offset
: is the distance from the physical edge of the paper to the place where all text begins. In normal-world terms, this distance is called the 'left margin'. Normally you only set the page-offset at the very start of a formatting job and you never change it again.

Line Length
: is the distance from the left margin (or page-offset) to the right edge of the text. The line-length is ***relative*** to the page-offset. In some respects, 'line-length' is a bit of a misnomer, because once you have set the page-offset at the start of the document (and assuming you never change it), the line-length really nails down the position of the ***right margin*** and has little to do with the length of the line.

Indent
: is where the left edge of your text starts. Normally the indent is zero, so that the edge of the text is where the page-offset is, but you can change the indent so that the text starts somewhere else. Note that the line-length is not affected by the indent — that is, indenting the text doesn't change the position of the right margin.

Page Length
: is the distance from the extreme top of the page to the extreme bottom of the page, that is, the page length is the physical length of the paper.

The figure below is a diagram of a page of text with the relevant parts pointed out. This diagram is a scale-model of an 8.5 × 11-inch sheet of paper, so while the numbers quoted in the text below are expressed in 'real' units, the actual dimensions are scaled.

left header center header right header

This paragraph has the *page-offset* set to give a left margin of approximately one inch (scaled). The *line-length* is set to 6.5 inches (scaled). This means there is a one-inch (scaled) left margin and a one-inch (scaled) right margin. The *indent* is set to zero so that the *current left margin* is at the same place as the page-offset.

> This paragraph has the *page-offset* and the *line-length* the same as the last paragraph, but we've used a **.in +0.5i** request to *indent* the left margin by half an inch — the *current left margin* is now page-offset+indent. Note that the position of the *right margin* remains the same as in the previous paragraph — only the left margin moved, so the effective length of the lines is shorter.

This paragraph now has the left margin back to the original position because we inserted a **.in -0.5i** request before it.

This paragraph has the left margin moved, not by indenting, but by changing the *page-offset*. via a **.po +0.5i** request. Now *all* text is moved to the left and because the line-length hasn't changed, the right margin moves as well.

This is the regular old paragraph where the first line is indented and the rest of the text in the paragraph is flushed to the left margin. The first line was indented via a **.ti +0.25i** request to give a *temporary indent* of the first line.

- This paragraph is an example of an 'item' or 'bulletted' or 'hanging' paragraph, where the left margin is moved to the right, and the 'bullet' or 'tag' is moved back to the old left margin. This effect was achieved via a **.in +0.25i** request to move the left margin rightwards, and then the 'bullet' was preceded by a **.ti -0.25i** request to get a *temporary indent* to the old position of the left margin.

Finally, you should note that *tab stops* are *relative* to the *current left margin* as we shall show with a couple of blocks of text with different indents. Note that the positions of the tab stops are shown with ! characters:
! ! ! ! ! !
You can see by the line of ! marks above where the tab stops are.

> Now we have another block of text here but with the indent moved over a half-inch. As you can see by the line of ! marks below, the tab stops have moved with the left margin:
> ! ! ! ! ! !

left footer center footer right footer

Figure 15-2: Layout of a Page

15.3.1. Margins and Indentations

As we said above, the positions of the left-hand and right-hand margins are controlled via the page-offset and the line-length. After that, any movements of the left-hand margin are controlled via indent and temporary indent requests. These topics are discussed in the following subsections.

15.3.1.1. **.po** — *Set Page Offset*

The ***page-offset*** is the distance from the extreme left-hand edge of the paper to the left margin of your text. When you use 'standard' 8.5X11-inch paper, it is customary to have the left and right margins be one inch each, so that the physical length of the printed lines are 6.5 inches — or you'd say that the measure was 39 picas if you're a typographer and can't handle inches.

In general, you only set the page-offset once in the course of formatting a document. Setting the page-offset determines the position of the physical left margin for the text, and then you (almost) never change the page-offset again — all indentation is done via **.in** (**in**dent) requests and **.ti** (**t**emporary **i**ndent) requests. We talk about these requests later in this part of the manual.

The position of the physical right margin for the text is determined by the ***line-length*** relative to the page-offset. The **.ll** (**l**ine **l**ength) request is discussed below.

Summary of the .po Request

Item	*Description*
Form of Request:	**.po** ±*N*
Initial Value:	0 in ***nroff***, 26
If No Argument:	Previous value
Explanation:	Set the current ***left margin*** to ±*N*. In ***troff*** the initial value is 26/27 inch, which provides about one inch of paper margin including the physical typesetter margin of 1/27 inch. In ***troff*** the maximum (line-length)+(page-offset) is about 7.54 inches. In ***nroff*** the initial page-offset is zero.
Notes:	**v**

The current page-offset is available in the **.o** register.

15.3.1.2. **.ll** — *Set Line Length*

troff gives you full control over the length of the printed lines. By the way, typographers don't use mundane phrases like 'line-length', they use the word 'measure' to mean the length of a line. They get confused if you talk inches or centimeters at them, instead they always talk in 'picas'.

Nevertheless, to set the line-length in ***troff***, use the **.ll** (**l**ine **l**ength) request, as in

```
.ll 6i
```

As with the **.sp** request, the actual length can be specified in several ways — inches are probably the most intuitive unless you live in one of the very few places in the world where they don't use inches.

The maximum line-length provided by the typesetter is 7.5 inches, by the way. 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 by the **.po** (**p**age **o**ffset) request discussed above.

```
.po 0
```

sets the offset as far to the left as it will go.

Note that the line-length ***includes*** indent space but ***not*** page-offset space. The line-length minus the indent is the basis for centering with the **.ce** request. The effect of the **.ll** request 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 is available in the **.l** number register. The length of three-part titles produced by a **.tl** request (see the section on ***Titles, Pages, and Numbering***) is independent of the line-length set by the **.ll** request — the length of a tree-part title is et by the **.lt** request.

Summary of the .ll Request

Item	*Description*
Form of Request:	**.ll** ±*N*
Initial Value:	6.5 inches
If No Argument:	Use previous value
Explanation:	Set the line-length to *N* where is the value of the line length, or an increment or decrement for the line-length. In ***troff*** the maximum (line-length)+(page-offset) is about 7.54 inches.
Notes:	E, **m**

15.3.1.3. **.in** — *Set Indent*

Given that you've got your page-offset and line-length correctly set for a document to establish the position of the left and right margins, you now make all other movements of the left margin via the **.in** (**in**dent) request discussed here, and via the **.ti** (**t**emporary **i**ndent) request described below.

The **.in** (**in**dent) request indents the left margin by some specified amount from the page-offset. This means that all the following text will be indented by the specified amount until you do something to change the indent. To get the first line of a paragraph indented, you don't use the **.in** request, but you use the **.ti** (**t**emporary **i**ndent) request described below.

As an example, a common text structure in books and magazines is the 'quotation' — a paragraph that is indented both on the right and the left of the line. A quotation is used for precisely that purpose, namely to set some text off from the rest of the copy. We can achieve such a paragraph by using the **.in** request to move the left margin in, and the **.ll** request to move the

right margin leftwards:

```
.in +0.5i
.ll -0.5i
I was to learn later in life that we tend to meet any new
situation by reorganizing; and a wonderful method
it can be for creating the illusion of progress
while producing confusion, inefficiency, and demoralization.
.ll +0.5i
.in -0.5i
```

When you format the above construct you get a block that looks like this:

> I was to learn later in life that we tend to meet any new situation by reorganizing; and a wonderful method it can be for creating the illusion of progress while producing confusion, inefficiency, and demoralization.[2]

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 +2i** makes lines two inches ***longer***, whereas **.ll 2i** makes them two inches ***long***:

```
.ll 2.0i
I was to learn later in life that we tend to meet any new
situation by reorganizing; and a wonderful method
it can be for creating the illusion of progress
while producing confusion, inefficiency, and demoralization.
```

> I was to learn later in life that we tend to meet any new situation by reorganizing; and a wonderful method it can be for creating the illusion of progress while producing confusion, inefficiency, and demoralization.

With **.in**, **.ll**, and **.po**, the previous value is used if no argument is specified.

Note that the line-length ***includes*** indent space but ***not*** page-offset space. The line-length minus the indent is the basis for centering with the **.ce** request. The effect of the **.in** request 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 indent is available in the **.i** number register.

---

[2] *Petronius Arbiter*, A.D. 60.

Summary of the .in Request

Item	*Description*
Form of Request:	**.in** ±*N*
Initial Value:	0
If No Argument:	Use previous value
Explanation:	Set the indent to ±*N* where *N* is the value of the indent, or an increment or decrement on the current value of the indent.
Notes:	E, **m**. The **.in** request causes a ***break***.

15.3.1.4. `.ti` — *Temporararily Indent One Line*

The **.ti** (temporary **i**ndent) request indents the ***next*** text line by a specified amount.

A common application for **.ti** is where the first line of a paragraph must be indented just like the one you're reading now. You get such a construct with a sequence like:

```
.ti 3
A common application for ...
          .
          .
          .
```

and when the paragraph is formatted, the first line of the paragraph is indented by three whatsits just like this one. Three of what? The default unit for the **.ti** request, as for most horizontally oriented requests — **.ll** (**l**ine **l**ength), **.in** (**in**dent), and **.po** (**p**age **o**ffset) — is ems. An ***em*** is roughly the width of the letter 'm' in the current point size. Thus, an em is always ***proportional*** to the point size you are using. An em in size ***p*** is the number of ***p*** points in the width of an 'm'. Here's an em followed by an em dash in several point sizes to show why this is a ***proportional*** unit of measure. You wouldn't want a 20-point dash if you are printing the rest of a document in 12-point text. Here's 12-point text:

m
|—|

Here's 16-point text:

m
|—|

And here's 20-point text:

m
|—|

Thus a temporary indent of **.ti 3** in the current point size results in an indent of three m's width or ¦mmm¦.

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 -0.3i
```

moves the next line back three tenths of an inch. A common text structure found in documents is 'itemized lists' where the paragraphs are indented but are set off by 'bullets' or some such. Item lists are often called 'hanging paragraphs' because the first line with the item on it 'hangs' to the left. For example, you could type the following series of lines like this (we've deliberately shortened the length of the line to illustrate the effects):

```
.ll 4.0i                                    shorten lines for this example
.in +0.2i                                   indent left margin by a fifth inch
.ta +0.2i                                   set a tab for the hanging indent
.ce                                         center a line of title
Indent Control Requests
.ti -0.2i                                   move left margin back temporarily
\(bu tab the .po request sets the
page-offset to the desired amount thereby making
sure the left margin is correct.
.ti -0.2i                                   move left margin back temporarily
\(bu tab the .in request sets the
indent from the left margin for all following text.
.ti -0.2i                                   move left margin back temporarily
\(bu tab the .ti request sets the indent for
the following line of text only thus providing for
fancy paragraph effects.
```

We had to play some tricks with tabs as well to get everything lined up, but that won't affect the main point of the discussion. The ***tab*** markers in the lines above show where there's a tab character, and the **\(bu** sequence at the start of the lines gets you a bullet (•) like that — we'll show the special character sequences later in this manual. When you format the text as shown in the example above, you get this effect:

Indent Control Requests

- the .po request sets the page-offset to the desired amount thereby making sure the left margin is correct.
- the .in request sets the indent from the left margin for all following text.
- the .ti request sets the indent for the following line of text only thus providing for fancy paragraph effects.

Note that the line-length ***includes*** indent space but ***not*** page-offset space. The effect of a **.ti** request 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 indent is available in the **.i** register.

Summary of the .ti Request

Item	*Description*
Form of Request:	**.ti** ±*N*
Initial Value:	0
If No Argument:	Ignored
Explanation:	Indent the ***next*** output text line a distance ±*N* with respect to the current indent. The resulting total indent may not be negative. The current indent is not changed.
Notes:	E, **m**. The **.ti** request causes a ***break***.

15.3.2. Page Lengths, Page Breaks, and Conditional Page Breaks

Neither ***nroff*** nor ***troff*** provide ***any*** facilities for top and bottom margins on a page, nor for any kind of page numbering at all. The **−ms** macro package described in a previous section of this manual sets things up so that reasonable pagination with top and bottom margins and page numbers are done automatically.

If you want top and bottom margins in when using raw ***troff*** or ***nroff***, you have to do some tricky stuff. The tricky stuff is done via ***traps*** and ***macros***. The ***trap*** tells ***troff*** or ***nroff*** ***when*** to do some processing for the margins (for example, you might set a trap to start the bottom margin 0.75 inches from the bottom of the page), and the ***macro*** defines ***what*** to do when the trap is sprung. It is conventional to set traps for them at vertical positions 0 (top) and −*N* (*N* from the bottom).

A pseudo-page 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 tables, 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).

The useable page width on the Graphic Systems phototypesetter is about 7.54 inches, beginning about 1/27 inch from the left edge of the 8 inch wide, continuous roll paper. The physical limitations on ***nroff*** output are output-device dependent.

15.3.2.1. **.pl** *— Set Page Length*

Just as the **.po**, **.ll**, **.in**, and **.ti** requests changed the horizontal aspects of the page, the **.pl** (page length) request determines the physical length of the page. In general you ***never*** need to use the **.pl** request because the standard setting is right for all but the most esoteric purposes.

Summary of the .pl Request	
Item	*Description*
Form of Request:	**pluf** $\pm N$
Initial Value:	11 inches
If No Argument:	11 inches
Explanation:	Set page length 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.
Notes:	**v**

15.3.2.2. **.bp** — *Start a New Page*

Summary of the .bp Request	
Item	*Description*
Form of Request:	**.bp** $\pm N$
Initial Value:	N=1
If No Argument:	Increment current page number by 1.
Explanation:	Eject the current page and start a new page. If $\pm N$ is given, the new page number will be $\pm N$. Also see the **.ns** (no space) request.
Notes:	**v**, The **.bp** request causes a ***break***.

15.3.2.3. **.pn** — *Set Page Number*

Summary of the .pn Request	
Item	*Description*
Form of Request:	**.pn** $\pm N$
Initial Value:	N=1
If No Argument:	Ignored
Explanation:	The next page (when it occurs) will have the page number $\pm N$. A **pn** must occur before the initial pseudo-page transition to affect the page number of the first page. The current page number is in the **%** register.

15.3.2.4. **.ne** — *Specify Space Needed*

In some applications you need to make sure that a few lines of text all appear together on the same page. There are several ways to acheieve this ranging from simple to complicated. One of the simplest ways is to use the **.ne** (**ne**ed) lines request:

```
.ne 3                    specify we need at least three lines
some
lines
of
text
to
be
kept
on the
same page
```

The arrangement of the **.ne** request specifies that if there are many lines of text in (say) a paragraph, at least three of the lines will appear together on the same page, otherwise a new page will be started. The object of this exercise is to avoid what typographers call 'orphans' — that is, the first line of a paragraph appearing all alone and lonely on the bottom of a page, while the rest of the paragraph appears on the next page. This is generally considered to be somewhat ugly and should be avoided if possible. By itself, ***troff*** is too stupid to recognize the existence of orphans (indeed of any text constructs at all), but the facilities are there to avoid these situations. In general, macro packages such as the **−ms** macro package discussed elsewhere have 'begin paragraph' macros such as **.PP** which take care of controlling orphans.

Summary of the .ne Request

Item	*Description*
Form of Request:	**.ne** *N*
Initial Value:	Not applicable
If No Argument:	1V
Explanation:	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 there are no remaining traps on 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 ***diversion trap***, if any, or is very large.
Notes:	**v**

15.3.3. Multi-Columnar Page Layout by Marking and Returning

It is possible to achieve multi-column output in ***troff*** or ***nroff*** via the **.mk** (mark) and **.rt** (return) requests. Other nifty special effects can also be obtained using these requests, but one of the common uses is to do multi-column output. Basically, the **.mk** request ***marks*** the current vertical position on the page (you can place the result of the mark in a register). You do a

column's worth of output, then when you get to the end of the page, instead of starting the next page, you return (via the **.rt** request) to the marked position, set up a new indent and line-length, and crank out another column.

15.3.3.1. **.mk** — *Mark Current Vertical Position*

Summary of the .mk Request

Item	*Description*
Form of Request:	**.mk** *R*
Initial Value:	Not applicable
If No Argument:	*R* is an internal register
Explanation:	Mark the ***current*** vertical place in an internal register (both associated with the current diversion level), or in register *R*, if given. See **rt** request.

15.3.3.2. **.rt** — *Return to Marked Vertical Position*

Summary of the .rt Request

Item	*Description*
Form of Request:	**.rt** $\pm N$
Initial Value:	**Not applicable**
If No Argument:	**place marked by a previous .mk request.**
Explanation:	Return ***upward only*** to a marked vertical place in the current diversion. If $\pm N$ (with respect to the 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 (refer to the section on *Line Spacing and Character Sizes*) may be used in all cases instead of **rt** by spacing to the absolute place stored in a explicit register; for example, using the sequence

15.4. Line Spacing and Character Sizes

15.4.1. **.sp** — *Get Extra Space*

You get extra vertical space with the **.sp** (**sp**ace) request. A simple

```
.sp
```

request with no argument 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 the **.vs** request to define line spacing, and in fact after most requests that deal with physical dimensions.

It should be noted that all size numbers are converted internally to 'machine units', which are 1/432 inch (1/6 point). For most purposes, this is enough resolution that you don't have to worry about the accuracy of the representation. The situation is not quite so good vertically, where resolution is 1/144 inch (1/2 point).

The vertical spacing *(V)* between the base-lines 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 11-point type with a 13-point vertical spacing. The current *V* is available in the **.v** register.

Summary of the .vs Request	
Item	***Description***
Form of Request:	**.vs** *N*
Initial Value:	1
If No Argument:	Use previous value
Explanation:	**Set vertical base-line spacing size** *V*. Transient *extra* vertical space available with **\x** ´*N* ´ (see above).
Notes:	E, **p**

15.4.2. **.ls** — *Change Line Spacing*

Multiple-*V* line separation (for instance, double spacing) can be requested with the **.ls** (line spacing) request.

Summary of the .ls Request	
Item	***Description***
Form of Request:	**.ls** *N*
Initial Value:	*N*=1
If No Argument:	Use previous value
Explanation:	Set ***line*** spacing to ±*N*. *N*−1 *Vs* ***(blank lines)*** are appended to each output text line. Appended blank lines are omitted, if the text or previous appended blank line reached a trap position.
Notes:	E

15.4.3. **\x** *Function — Get 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 **\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 used post-line extra line-space is available in the **.a** register.

15.4.4. **.vs** — *Change Vertical Distance Between Lines*

The other parameter that determines what the type looks like is the spacing between lines, which is set independently of the point size. Vertical spacing is measured from the bottom of one line to the bottom of the next. The bottom of the text on a line is often called the ***baseline***. The vertical spacing is often called ***leading*** (pronounced 'led-ing') and comes from the days when text was produced with lead slugs instead of electronic widgets like laser printers.

You control vertical spacing with the **.vs** (**v**ertical **s**pacing) request. For running text, it is usually best to set the vertical spacing about 20% bigger than the character size. For example, so far in this document, we have used 11-point type with a vertical line-spacing of 7 points between baselines. Typographers call this '11 on 7', so when you hear some one say that a book is set in '9 on 11', you know that it's 9-point type with 11-point vertical spacing.

So, somewhere at the start of this document, the macro package that formats this document for us had requests like:

```
.ps 11p
.vs 13p
```

Had we set the point size and the vertical spacing like this:

```
.ps 11p
.vs 11p
```

the running text would look like this. After a few lines, you will agree it looks a little cramped. The right 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 14.

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 size and vertical spacing respectively.

15.4.5. **.sp** — *Get Blocks of Vertical Space*

A block of vertical space is ordinarily requested using the **.sp** (**s**pace) request, which honors the ***no-space*** mode and which does not space ***past*** a trap. A contiguous block of vertical space may be reserved using the **.sv** request (see below).

Summary of the .sp Request	
Item	*Description*
Form of Request:	**.sp** *N*
Initial Value:	Not applicable
If No Argument:	*N*=1V
Explanation:	`Space vertically in` ***either*** direction. If *N* is negative, the motion is ***backward*** (upward) and is limited to the distance to the top of the page. Forward (downward) 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).
Notes:	B, **v**

15.4.6. **.sv** — *Save Block of Vertical Space*

Summary of the .sv Request	
Item	*Description*
Form of Request:	**.sv** *N*
Initial Value:	Not applicable
If No Argument:	*N*=1V
Explanation:	Save a contiguous vertical block of size *N*. If the distance to the next trap is greater than *N*, *N* vertical space is output. No-space mode has *no* effect. If this distance is less than *N*, no vertical space is immediately output, but *N* is remembered for later output (see the **.os** request). Subsequent **sv** requests will overwrite any still remembered *N*.
Notes:	**v**

15.4.7. **.os** — *Output Saved Vertical Space*

Summary of the .os Request	
Item	*Description*
Form of Request:	**.os**
Initial Value:	Not applicable
If No Argument:	Not applicable
Explanation:	Output saved vertical space. No-space mode has *no* effect. Used to finally output a block of vertical space requested by an earlier **sv** request.

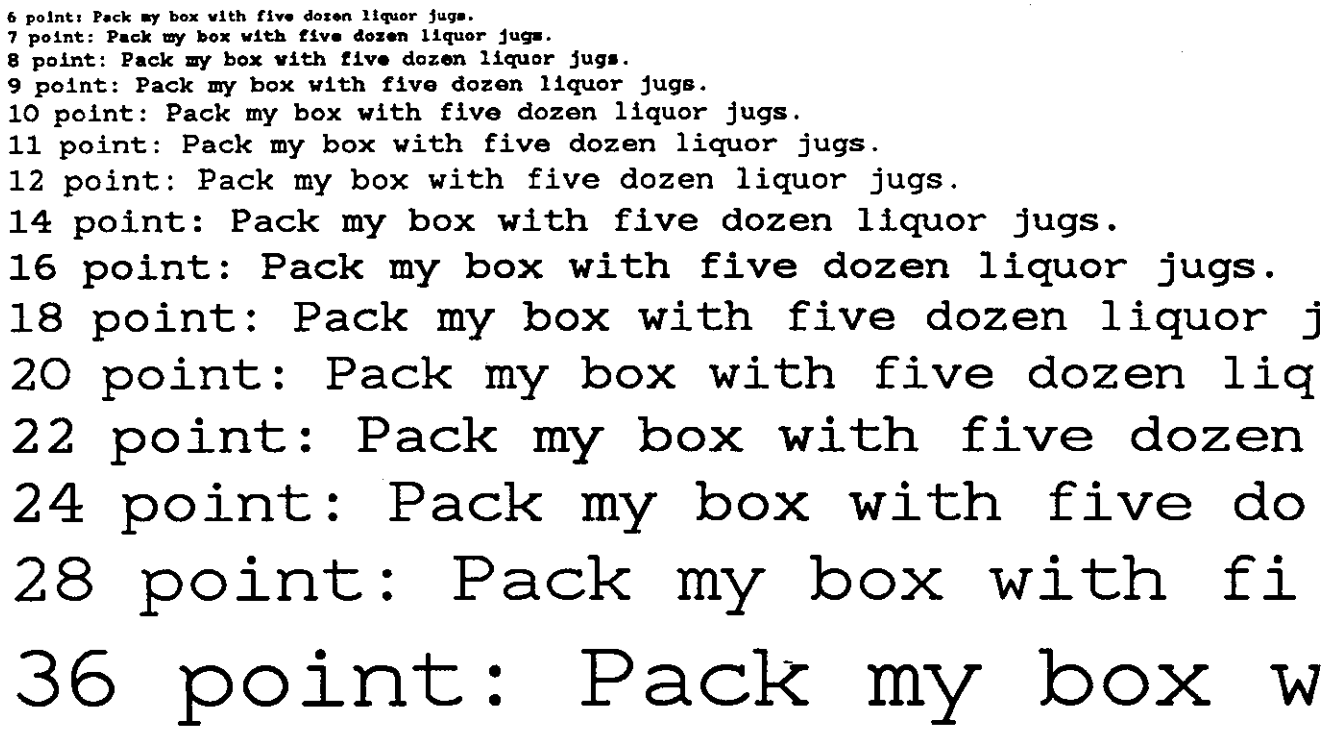
15.4.8. **.ns** — *Set No Space Mode*

Summary of the .ns Request	
Item	*Description*
Form of Request:	**.ns**
Initial Value:	Not applicable
If No Argument:	Not applicable
Explanation:	Turn on no-space mode — 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**.
Notes:	D

Summary of the .rs Request	
Item	*Description*
Form of Request:	**.rs**
Initial Value:	Not applicable
If No Argument:	Not applicable
Explanation:	Restore spacing — turn off no-space mode.
Notes:	D

15.4.9. **.ps** — *Change the Size of the Type*

In *troff*, you can change the physical size of the characters that are printed on the page. The **.ps** (**p**oint **s**ize) request 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 1/2 inch. *Troff* and the machine it was originally designed for understand 15 point sizes, listed below.



If the number after a **.ps** request is not one of these legal sizes, it is rounded up to the next valid value, with a maximum of 36. If no number follows **.ps**, *troff* reverts to the previous size, whatever it was. *troff* begins with point size 10, which is usually fine. This document is in 11-point.

The point size can also be changed in the middle of a line or even a word with an in-line size change sequence. In general, text which is in ALL CAPITALS in the middle of a sentence tends to loom large over the rest of the text and so it is customary to drop the point size of the capitals so that it looks like ALL CAPITALS instead. You use the **\s** (for **s**ize) sequence to state what the point size should be. You can state the size explicitly as in this line here:

```
The \s8POWER\s0 of a \s8SUN\s0
```

to produce the output line like:

```
The POWER of a SUN
```

As above, **\s** should be followed by a legal point size, except that **\s0** makes the size revert to its previous value (before you just changed it).

Note that because there are a fixed number of point sizes that the system knows about, the sequence **\s96** gets you a nine-point 6 instead of 96-point type like you wanted, whereas the sequence **\s180** gets you an 18-point 0 instead of 180-point type.

Stating the point size in absolute terms as above is not always a good idea — what you ***really*** want is for the changed size to be ***relative*** to the surrounding text, so that if your document is in 11-point type like this one, you'd really like the bigger (or smaller stuff) to be a couple of points different without your having to know explicitly what the actual size is. So in this case, you can use a relative size-change sequence of the form **\s+*n*** to raise the point size, and **\s−*n*** to lower the point size. The number ***n*** is restricted to a ***single digit***. So we can rework our previous example from above like this:

```
The \s-2POWER\s+2 of a \s-2SUN\s+2
```

to produce the output line like:

```
The POWER of a SUN
```

Relative size changes have the advantage that the size difference is independent of the starting size of the document. Of course this stuff only works really well (in typography terms) when the changes in size aren't too violently out of whack with the point size — a change of two points in 36-point type doesn't have quite the same impact as it does for 12-point type — there is a question of the ***weight*** of the type, but by the time you get to that stuff you'll be much more knowledgeable about typography.

Character point sizes available on the Graphic Systems typesetter are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36. This is a 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 between any two characters by embedding a **\s***N* at the desired point to set the size to *N*, or a **\s**±*N* ($1 \leq N \leq 9$) to increment/decrement the size by *N*; **\s0** restores the ***previous*** size. Requested point size values that are between two valid sizes yield the larger of the two. The current size is available in the **.s** register. ***nroff*** ignores type size control.

Summary of the .ps Request

Item	*Description*
Form of Request:	**.ps** ±*N*
Initial Value:	10 points
If No Argument:	Use previous value
Explanation:	Set point-size to ±*N*. Alternatively embed **\s***N* or **\s**±*N*. 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* works because the previous requested value is also remembered. Ignored in ***nroff***.
Notes:	E

15.4.10. **.ss** — *Set Size of Space Character*

Summary of the .ss Request	
Item	***Description***
Form of Request:	**.ss** *N*
Initial Value:	12
If No Argument:	Ignored
Explanation:	Set space-character size to *N*/36 ems. This size is the minimum word spacing in adjusted text. Ignored in *nroff*.
Notes:	E

15.4.11. **.cs** — *Set Constant Width Characters*

Summary of the .cs Request	
Item	***Description***
Form of Request:	**.cs** *F N M*
Initial Value:	Off
If No Argument:	Ignored
Explanation:	Constant character space (width) mode is set on for font *F* (if mounted); the width of every character is taken as *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 *F* are also so treated. If *N* 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*.
Notes:	P

15.5. Fonts and Special Characters

troff and the typesetter allow four different fonts at any one time. Normally three fonts (Times roman, italic and bold) and one collection of special characters are permanently mounted.

abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

The greek, mathematical symbols and miscellany of the special font are listed in Appendix A.

Troff prints in Roman unless told otherwise. To switch into bold, use the **.ft** (**font**) request:

```
.ft B
```

and for italics,

```
.ft I
```

To return to roman, use **.ft R**; to return to the previous font, whatever it was, use either **.ft P** or just **.ft**. The 'underline' request

```
.ul
```

makes the next input line print in italics. **.ul** 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 word with the in-line request **\f**:

bold*face* text

is produced by

```
\fBbold\fIface\fR text
```

If you want to do this so the previous font, whatever it was, is left undisturbed, insert extra in-line **\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 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 **.fp** (**font position**) request tells *troff* what fonts are physically mounted on the typesetter:

```
.fp 3 H
```

says that the Helvetica font is mounted on position 3. Appropriate **.fp** requests should appear at the beginning of your document if you do not use the standard fonts.

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 (R) on 1, italic (I) on 2, bold (B) on 3, and special (S) on 4 — remember 'RIBS'.

There is also a way to get 'synthetic' bold fonts by overstriking letters with a slight offset. Look at the **.bd** request.

Special characters have four-character names beginning with **\(**, and they may be inserted anywhere. For example,

¼ + ½ = ¾

is produced by

```
\(14 + \(12 = \(34
```

In particular, greek letters are all of the form **\(***—, where — is an upper or lower case roman letter reminiscent of the greek. Thus to get

$\Sigma(\alpha \times \beta) \rightarrow \infty$

in bare *troff* we have to type

```
\(*S(\(*a\(mu\(*b) \(-> \(if
```

That line is unscrambled as follows:

Escape Sequence	*Character Printed*	*Description*
\(*S	Σ	*Upper-case Sigma or Sum*
(	(	
\(*a	α	*lower-case alpha*
\(mu	×	*multiplication sign or signum*
\(*b	β	*lower-case beta*
)	)	
\(->	→	*tends towards*
\(if	∞	*infinity*

A complete list of these special names occurs in Appendix A.

In *eqn* (*Formatting Mathematics with 'eqn'*) the same effect can be achieved with the input

```
SIGMA ( alpha times beta ) -> inf
```

which is less concise (31 keystrokes instead of 27!), but clearer to the uninitiated.

Notice that each four-character name is a single character as far as *troff* is concerned — the 'translate' request

```
.tr \(mi\(em
```

is perfectly clear, meaning

.tr − —

that is, to translate − (minus sign) into — (em-dash).

Some characters are automatically translated into others: grave ` and acute ´ accents (apostrophes) become open and close single quotes ‘ ’; the combination of “...” is generally preferable to the double quotes "...". Similarly a typed minus sign becomes a hyphen -. To print an explicit − sign, use **\-**. To get a backslash printed, use **\e**.

15.5.1. Character Set

The *troff* character set consists of the Graphics Systems Commercial II character set plus a Special Mathematical Font character set—each having 102 characters. These character sets are shown in Appendix A, *Examples of Fonts and Non–ASCII Characters*. 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 **\(***xx* where *xx* is a two-character name also explained in Appendix A. The three ASCII exceptions are mapped as follows:

Table 15-6: Exceptions to the Standard ASCII Character Mapping

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 **\´**, **\`**, and **\−** respectively or by their names found in Appendix A. The ASCII characters @, #, ", ´, `, <, >, \, {, }, ~, ^, 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 in general print only ASCII characters, additional characters as may be available on the output device, such characters as may be constructed by overstriking or other combination, 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 characters ´, `, and _ print as themselves.

15.5.2. Fonts

The default mounted fonts are Times Roman (**R**), Times Italic (**I**), Times Bold (**B**), and the Special Mathematical Font (**S**) on physical typesetter positions 1, 2, 3, and 4 respectively. These fonts are used in this document. The *current* font, initially Roman, may be changed (among the mounted fonts) by use of the **ft** request, or by embedding at any desired point either **\f***z*, **\f(***zz*,

or \f*N* 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 automatically handled. A request for a named but not-mounted font is *ignored*. *troff* can be informed that any particular font is mounted by use of the **fp** request. The list of known fonts is installation dependent. 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**.

nroff understands font control and normally underlines Italic characters.

15.5.3. **.bd** — *Artificial Bold Face*

Summary of the .bd Request

Item	*Description*
Form of Request:	**.bd** *F N*
Initial Value:	Off
If No Argument:	No Emboldening
Explanation:	Artificially embolden characters in font *F* by printing each one twice, separated by *N*−1 basic units. A reasonable value for *N* is 3 when the character size is in the vicinity of 10 points. If *N* is missing the embolden mode is turned off. The mode must be still or again in effect when the characters are physically printed. Ignored in *nroff*.
Form of Request:	**.bd S** *F N*
Explanation:	Embolden characters in the Special Font whenever the current font is *F*. The mode must be still or again in effect when the characters are physically printed.
Notes:	P

15.5.4. **.ft** — *Set Font*

Summary of the .ft Request

Item	*Description*
Form of Request:	**.ft** *F*
Initial Value:	Roman
If No Argument:	Previous Font
Explanation:	Change font to *F*. Alternatively, embed \f*F*. The font name **P** is reserved to mean the previous font.
Notes:	E

15.5.5. **.fp** — *Set Font Position*

Summary of the .fp Request	
Item	*Description*
Form of Request:	**.fp** *N F*
Initial Value:	R, I, B, S
If No Argument:	Ignored
Explanation:	Font position — this is a statement that a font named *F* is mounted on position *N* (1-4). It is a fatal error if *F* is not known. The phototypesetter has four fonts physically mounted. Each font consists of a film strip which can be mounted on a numbered quadrant of a wheel. The default mounting sequence assumed by *troff* is R, I, B, and S on positions 1, 2, 3 and 4. Any **fp** request specifying a font on some position must precede **fz** requests relating to that position.

15.5.6. **.fz** — *Force Font Size*

Summary of the .fz Request	
Item	*Description*
Form of Request:	**.fz** *S F N*
Initial Value:	None
If No Argument:	None
Explanation:	Forces font *F* or *S* for special characters to be in size *N*. A **.fz 3 −2** causes implicit \s−2 every time font 3 is entered, and a matching \s+2 when left. Same for Special font characters that are used during *F*. Use *S* to handle Special characters during *F* (**fz 3 −3** or **fz S 3 −0** cause automatic reduction of font 3 by 3 points while special characters are not affected. Any **fp** request specifying a font on some position must precede **fz** requests relating to that position.

15.5.7. **.lg** — *Control Ligatures*

Ligatures are not something you sprain when you work out too hard, rather a ligature is a special way of joining two characters together as one. Way back in the days before Gutenberg, scribes would have a variety of special forms to choose from to make lines come out all the same length on a manuscript. Some of these forms are still with us today. Five ligatures are available in the current *troff* character set — **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 during input.

If you want other ligatures like the æ, œ, Æ, andŒ ligatures, you have to make them up yourself — ***troff*** doesn't know about them. See the section on ***Arbitrary Horizontal Motion with the \h Function*** for some examples on constructing these ligatures.

Summary of the .lg Request

Item	*Description*
Form of Request:	**.lg** *N*
Initial Value:	Off in ***nroff***, on in ***troff***.
If No Argument:	on
Explanation:	Turn Ligature mode on if *N* is absent or non-zero. Turn ligature mode off if *N*=0. If *N*=2, 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 ***nroff***.

15.6. Tabs, Leaders, and Fields — Aligning Things in Columns

There are several ways to get stuff lined up in columns, and to achieve other effects such as horizontal motion and repeated strings of characters. The three related topics we discuss in this section are ***tabs***, ***leaders***, and ***fields***.

tabs behave just like the tab stops on a typewriter.

leaders are for generating repeated strings of characters.

fields are a general mechanism for helping to line stuff up into columns.

This part of the document concentrates on the 'easy' parts, so to speak. Later sections of this document contain discussions on the facilities for drawing lines and for producing arbitrary motions on the page.

15.6.1. `.ta` — *Set Tabs*

Tabs (the ASCII 'horizontal tab' character) can be used to produce output in columns, or to set the horizontal position of output. Typically tabs are used only in unfilled text. Tab stops are set by default every half inch from the current indent (in ***troff***) and every 0.8 inch from the current indent (in ***nroff***), but can be changed by the **.ta** (**ta**b) request. In the example below, we set tab stops every one-and-a-half inches and set some text in columns based on those tab stops. We place a line of ! marks above and below the text to show where the tabs stops are in the output page:

```
.ta 1.5i 3.0i 4.5i 6.0i                          set tabs
!tab!tab!tab!tab!                                show where tabs are with ! character
word-one tab word-two tab word-three tab word-four tab word-five
!tab!tab!tab!tab!                                show where tabs are with ! character
```

When we format the above example, we get this output:

!	!	!	!	!
word-one	word-two	word-three	word-four	word-five
!	!	!	!	!

15.6.1.1. Setting Relative Tab Stops

The tab stops set in the example above are in terms of ***absolute*** position on the line. You could also set tabs ***relative*** to previous tabs stops by preceding the tab stop number with a + sign, and get exactly the same result:

```
.ta 1.5i +1.5i +1.5i +1.5i                       set tabs
!tab!tab!tab!tab!                                show where tabs are with ! character
word-one tab word-two tab word-three tab word-four tab word-five
!tab!tab!tab!tab!                                show where tabs are with ! character
```

15.6.1.2. Right-Adjusted Tab Stops

In the standard case as shown in the above examples, the tab stops are left-adjusted (as on a typewriter). You can also make the tab stops right-adjusting for doing things like lining up columns of numbers. When you right-adjust a tab stop, the action of placing a tab before the field places the material ***behind*** the tab stop on the output line. Here's an example of some input with both alphabetic and numeric items:

```
.nf
.ta 2.0iR
Julytab5
Augusttab9
Septembertab15
Octobertab60
Novembertab85
Decembertab126
.fi
```

Notice the **.ta** request — it has the letter **R** on the end to indicate that this is a right-adjusted tab. When we format that table, we get this result:

July	5
August	9
September	15
October	60
November	85
December	126

Notice how the numbers in the second column line up.

15.6.1.3. Centered Tab Stops

Finally you can make a ***centered*** tab stop, so that things get centered between the tabs. We can use the centering tabs to put a title on our table from above:

```
.nf
.ta 2.0iC
MonthtabShipments
.ta 2.0iR
Julytab5
Augusttab9
Septembertab15
Octobertab60
Novembertab85
Decembertab126
.fi
```

and when we format this table now, we get this result:

Month	Shipments
July	5
August	9
September	15
October	60
November	85
December	126

Notice that the column headings are centered over the data in the table.

If you have a complex table, instead of using *troff* or *nroff* directly, use the *tbl* program described in the chapter *Formatting Tables with 'tbl'*. A good example of where *tbl* does more work for you is when numerically-aligned items have decimal points in them — it is really hard to do this using the raw *troff* or *nroff* capabilities.

15.6.1.4. `.tc` — *Change Tab Replacement Character*

A tab inserts blank space between the item that came before and after it. You can change this by filling up tabbed-over space with some other character. Set the 'tab replacement character' with the **.tc** (**t**ab **c**haracter) request:

```
.ta 2.5i 4.5i
.tc _
Name tab Age tab
```

This produces

Name _____ Age _____

There is a more general mechanism for drawing lines, described in the section *Drawing Vertical and Horizontal Lines*.

To reset the tab replacement character to a space, use the **.tc** request with no argument. Lines can also be drawn with the in-line **\l** command, described in the section *Arbitrary Motions and Drawing Lines and Characters*.

Summary of the .tc Request

Item	*Description*
Form of Request:	**.tc** *c*
Initial Value:	space
If No Argument:	Removed
Explanation:	The tab repetition character becomes *c*, or is removed, specifying motion.
Notes:	E

15.6.1.5. Summary of Tabs

The table below is a summary of the types of tab stops. 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 was found) to the next tab stop
next-string	consists of the input characters following the tab up to the next tab or end of line
W	is the width of ***next-string***.

Table 15-7: Types of Tab Stops

Tab letter	*Tab type*	*Length of motion or repeated characters*	*Location of next-string*
blank	Left	*D*	Following *D*
R	Right	*D*−*W*	Right adjusted within *D*
C	Centered	*D*−*W*/2	Centered on right end of *D*

Summary of the .ta Request

Item	*Description*
Form of Request:	**.ta** *Nt ...*
Initial Value:	0.8inches in ***nroff***, 0.5 inches in ***troff***.
If No Argument:	Ignored
Explanation:	Set tab stops and types — *N* is the tab stop value and *t* is the type. ***troff*** tab stops are preset every 0.5 inches; ***nroff*** tab stops are preset every 0.8 inches. *t*=**R** means right-adjusting tabs, *t*=**C** means centering tabs, and if *t* is absent, the tabs are left-adjusting tab stops. Stop values in the list of tab stops are separated by spaces, and a value preceded by + is treated as an increment to the previous stop value.
Notes:	E, **m**

15.6.2. Leaders — Repeated Runs of Characters

Leaders are repeated runs of the same character between tab stops. Leaders are most often used to hang two separated pieces of text together. A common application is in tables of contents. If you look at the table of contents for this manual you will see that the chapter and section titles (on the left of the line) are separated from the page number (on the right end of the line) by a row of dots. In fact here is a short example to illustrate what the leaders look like:

Table of Contents

2.0 Blunt Uses of Clubs 13
2.1 Social Clubs 16
2.2 Arthritic Clubs 18
2.3 Golf Clubs 25
2.4 Two-by-Four Clubs 29

The dots are called ***leaders***, because they 'lead' your eye from one thing to the other. It is not nearly so easy to read stuff like that if the leaders aren't there:

Table of Contents

2.0 Blunt Uses of Clubs 13
2.1 Social Clubs 16
2.2 Arthritic Clubs 18
2.3 Golf Clubs 25
2.4 Two-by-Four Clubs 29

The leader character is normally a period, but it can in fact be any character you like — some people prefer dots and some people prefer a straight line:

Table of Contents

2.0 Blunt Uses of Clubs _____ 13
2.1 Social Clubs _____ 16
2.2 Arthritic Clubs _____ 18
2.3 Golf Clubs _____ 25
2.4 Two-by-Four Clubs _____ 29

A leader is very similar to a tab, but you get the repeated characters by typing an in-line **\a** sequence instead of a tab or a **\t** sequence. The **\a** sequence is a control-A character or an ASCII SOH (start of heading) character and is hereafter known as the ***leader*** character for the purposes of this discussion. When the leader character is encountered in text it generates a string of repeated characters. The length of the repeated string of characters is governed by internal ***tab stops*** specified just as for ordinary tabs as discussed in the section on tabs above.. The major difference between tabs and leaders is that tabs generate ***motion*** and leaders generate a ***string of periods***. Let's look at a fragment of the text that generated the examples above:

```
.ta 5.0i-5nR 5.0iR
2.0  Blunt Uses of Clubs  \a\t13"
  2.1  Social Clubs  \a\t16"
  2.2  Arthritic Clubs  \a\t18"
  2.3  Golf Clubs  \a\t25"
  2.4  Two-by-Four Clubs  \a\t29"
```

What we're trying to get here are lines of text with the section numbers and the titles, followed by a string of leader characters, followed by some space and then the page number at the right-hand end of the line. Tables of contents tend to look better with shorter line lengths, so we set

our first tab to five inches minus five en-spaces to leave a gap at the end of the leader. The second tab is set to a right-adjusting tab at five inches. Each line of the table now contains the text to appear on the left end, followed by a couple of spaces, followed by the **\a** sequence to indicated the leader, followed by the **\t** sequence to indicate the tab, and finally followed by the page number. The result of formatting all that stuff is:

2.0 Blunt Uses of Clubs 13
2.1 Social Clubs 16
2.2 Arthritic Clubs 18
2.3 Golf Clubs 25
2.4 Two-by-Four Clubs 29

15.6.2.1. **.lc** — *Change the Leader Character*

Just as you could use the **.tc** request to change the character that gets generated with tabs, you can use the **.lc** (leader character) request to specify the character that is generated by a leader. The standard leader character is the period. We can show this by taking our last fragment and placing a **.lc** request before it to change the leader character to an underline:

```
.lc _                              set leader character
.ta 5.0i-5nR 5.0iR                 set tabs
2.0  Blunt Uses of Clubs  \a\t13"
  2.1  Social Clubs  \a\t16"
  2.2  Arthritic Clubs  \a\t18"
  2.3  Golf Clubs  \a\t25"
  2.4  Two-by-Four Clubs  \a\t29"
```

Then when we format the thing, it looks like this:

2.0 Blunt Uses of Clubs _____ 13
2.1 Social Clubs _____ 16
2.2 Arthritic Clubs _____ 18
2.3 Golf Clubs _____ 25
2.4 Two-by-Four Clubs _____ 29

Whereas the length of generated motion for a tab can be negative, the length 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***.

Summary of the .lc Request

Item	*Description*
Form of Request:	**.lc** *c*
Initial Value:	.
If No Argument:	Removed specifying motion
Explanation:	The leader repetition character becomes *c*, or is removed specifying motion.
Notes:	E

15.6.3. **.fc** — *Set Field Characters*

A ***field*** is a more general mechanism for laying out material between tab stops. Hardly anyone ever needs to use fields, but the ***tbl*** preprocessor uses them for placing tabular material on the page. This section is a very short discussion on how to use fields. In general, when you want to lay out tabular material you should use ***tbl*** to do the job for you. Fields are a way of reducing the number of tab stops you have to set, and also have ***troff*** or ***nroff*** do some automatic work in parcelling out padding space for you.

A ***field*** lives between the current position on the ***input*** line and the next tab stop. The start and end of the field are indicated by a ***field delimiter*** character. ***troff*** or ***nroff*** places the field on the line and pads out any excess space with spaces. You indicate where the padding actually goes by placing ***padding indicator*** characters at various places in the field. You set the field delimiter character and the padding indicator character with the **.fc** (**f**ield **c**haracters) request. In the absence of any other information, ***troff*** or ***nroff*** has the field mechanism turned off entirely. The **.fc** request looks like:

```
.fc d p
```

where *d* is the field delimiter character and *p* is the padding indicator character. If you do not specify any character for a padding indicator, the ***space*** character is the default. However, this means that you could not have spaces within the field, so you normally specify the padding indicator as something other than a space.

So let's start with a ***very*** simple example of a single field and see what we get. Here is the input:

```
.ta 3.0i                          set a single tab at three inches
.fc # @                     set field delimiter character to # and
                                  set padding indicator character to @
!tab!                             the ! characters show where tabs are
#string of characters#
!tab!                             the ! characters show where tabs are
.fc
```

and here is the output after formattting:

```
!                                        !
string of characters
!                                        !
```

This is not very exciting — the characters in the field are simply left-adjusted in the field, and the rest of the field up to the tab stop are padded with spaces. You would get exactly the same result if you placed the padding indicator character at the right end of the field to indicate that you wanted the padding on the right:

```
.ta 3.0i                                 set a single tab at three inches
.fc # @                                  set field delimiter character to #
                                            set padding indicator character to @
&!tab!                                      the ! characters show where tabs are
#string of characters@#
!tab!                                       the ! characters show where tabs are
.fc
```

As you can see, the result is identical to the one just above:

```
!                                        !
string of characters
!                                        !
```

But now we can place a padding indicator character at the left end of the field and get strings right-adjusted in the field:

```
.ta 3.0i                                    set a single tab at three inches
.fc # @                                  set field delimiter character as #
                                            set padding indicator character as @
!tab!                                       the ! characters show where tabs are
#@string of characters#
#@another string of characters#
!tab!                                       the ! characters show where tabs are
.fc
```

We used two strings of different length here to show how they are right-adjusted against the tab stop:

```
!                                        !
                              string of characters
                      another string of characters
!                                        !
```

You can see how the spaces were placed on the left end of the field because that is we where we placed the padding indicator character, and the strings got adjusted right to the tab stop.

Then we can get fields ***centered*** by placing the padding indicator character at both ends of the string:

```
.ta 3.0i                                    set a single tab at three inches
.fc # @                                  set field delimiter character as #
                                                set padding indicator character as @
!tab!                                           the ! characters show where tabs are
#@string of characters@#
#@longer string of characters@#
!tab!                                           the ! characters show where tabs are
.fc
```

Again we used two strings of different lengths to show the effect of centering the field:

```
!                                       !
                string of characters
           longer string of characters
!                                       !
```

In general, a field or a sub-field *between* a pair of padding indicator characters is *centered* in its space on the line.

Things get even more useful when you have multiple sub-fields in a field — the padding spaces are then parcelled out so that the sub-fields are uniformly left-adjusted, right-adjusted, or centered between the current position and the next tab stop:

```
.ta w.0i                    set a single tab at five inches
.fc # @                     set field delimiter character as #
                            set padding indicator character as @
!tab!                       use the ! characters to show where tabs are
#string of characters#
#string of characters@another string#
!tab!                       use the ! characters to show where tabs are
```

and here is the output after we format that:

```
!                                                                       !
string of characters
string of characters                                        another string
!                                                                       !
```

And finally we can show three strings within a field, with the left part left-adjusted, the center part centered, and the right part right-adjusted:

```
.ta w.0i
.fc # @
!tab!
#left string@center string@right string#
#longer left string@longer center string@longer right string#
!tab!
```

and here is the output after we format that:

```
!                                                                       !
left string                  center string                  right string
longer left string      longer center string          longer right string
!                                                                       !
```

So to summarize, a ***field*** is contained between a ***pair*** of ***field delimiter*** characters. A field consists of sub-fields 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-fields and the field length is incorporated as horizontal padding space that is divided among the indicated padding places. The incorporated padding can be negative.

Summary of the .fc Request

Item	*Description*
Form of Request:	**.fc** *f p*
Initial Value:	Field mechanism off
If No Argument:	Field mechanism off
Explanation:	Set the field delimiter to *a*; set the padding indicator to *b* (if specified) or to the ***space*** character if *b* is not specified. In the absence of arguments, the field mechanism is turned off.

15.7. Titles, Pages, and Numbering

This is an area where things get tougher, because nothing is done for you automatically. Of necessity, some of this section is a cookbook, to be copied literally until you get some experience.

Suppose you want a title at the top of each page, saying just

left top center top right top

There was a very early text formatter called *roff*, where you could say

```
.he 'left top'center top'right top'
.fo 'left bottom'center bottom'right bottom'
```

to get headers and footers automatically on every page. Alas, this doesn't work in *troff*, which is a serious hardship for the novice. Instead you have to do a lot of specification:

- You have to say *what* the actual title is (reasonably easy — you just use the **.tl** request to specify what the title is).
- You have to specify *when* to print the title (also reasonably easy — you set a *trap* to call a *macro* that actually does the work),
- and finally you have to say *what* to do at and around the title line (this is the hard part).

Taking these three things in reverse order, first we define a **.NP** macro (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' request **'bp**, which skips to top-of-page (we'll explain the **'** shortly). Then we space down half an inch (with the **'sp**0.5i request), and print the title (the use of **.tl** should be self explanatory — later we will discuss parameterizing the titles), space another 0.3 inches (with the **'sp**0.3i request), 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' request **.wh**:

```
.wh  -1i  NP
```

No '.' is used before NP because 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** request appears in the input outside the definition of **.NP**; typically the input would be

```
.de NP
definition of the NP macro
..
.wh -1i NP
```

Now what happens? As text is actually being output, ***troff*** keeps track of its vertical position on the page. After a line is printed within one inch from the bottom, the **.NP** macro is activated. In the jargon, the **.wh** request sets a *trap* at the specified place, which is 'sprung' when that point is passed. **.NP** skips 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 **.bp** and **.sp**, and like several other requests, *break* the current line — 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 **.bp** or **.sp** in the **.NP** macro, a break would occur in the middle of the current output line when a new page is 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, something like this:

```
last line but one at almost at the bottom of the page
last line at the bottom of the
```

title on the bottom of the page

page break

title on the top of the next page

```
page.
```

This is ***not*** what we want. Using **'** instead of **.** for a request 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 list of requests that break lines is short and natural:

Table 15-8: Requests that Cause a Line Break

Command	*Explanation*
.bp	Begin a new page
.br	Break the current output line
.ce	Center line(s)
.fi	Start filling text lines
.nf	Stop filling text lines
.sp	Space vertically
.in	Indent the left margin
.ti	Temporary indent the left margin for the next line only

No other requests break lines, regardless of whether you use a **.** or a **'**. If you really ***do*** need a break, add a **.br** (**br**eak) request at the appropriate place.

One other thing to beware of — if you're changing fonts or point sizes a lot, 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 what you intended. Furthermore, the length of a title is independent of the current line length, so titles will come out at the default length of 6.5 inches unless you change it, which is done with the **.lt** (length of title) request.

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** request contain size or font changes. What we would like to do in cases like this is remember the status of certain aspects of the environment, change them to meet our needs for the time being, and then restore them after we're done with the special stuff. This requirement is satisfied by ***troff***'s 'environment' mechanism. discussed in the section on *Saving State With Environments*.

To get a footer at the bottom of a page, you can modify **.NP** so it does some processing ***before*** the **'bp** request, or split the job so that there is a separate footer macro invoked at the bottom margin and a header macro invoked at the top of the page.

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 skip to the new page. Again, **.bp +n** sets the page number to **n** more than its current value; **.bp** means **.bp +1**.

15.7.1. Three Part Titles for Running Headers and Footers

The **.tl** (title) request automatically places three text fields at the left, center, and right of a line (with a title-length specifiable via the **.lt** (length of title) request. The most common use for three-part titles is to put running headers and footers at the top and bottom of pages just like those in this manual. In fact, the **.tl** request may be used anywhere, and is independent of the normal text collecting process. For example, we just placed a three-part title right here in the text:

by typing the a three-part title request that looks like:

.tl 'Hunting the Snark'– % –'Smiles and Soap'

and you might notice that the page number in the formatted example is the same as the page number for this page.

Summary of the .tl Request

Item	*Description*
Form of Request:	**.tl** *'left 'center 'right '*
Initial Value:	Nothing
If No Argument:	Nothing
Explanation:	The strings ***left***, ***center***, and ***right*** 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.

Summary of the .pc Request

Item	*Description*
Form of Request:	**.pc** *c*
Initial Value:	**%**
If No Argument:	Off
Explanation:	Set the page number character to *c*, or remove it if there is no *c* argument. The page-number ***register*** remains **%**.

Summary of the .lt Request

Item	*Description*
Form of Request:	**.lt** *c*
Initial Value:	6.5 inches
If No Argument:	Use previous value
Explanation:	Set length of title to $\pm N$. The line-length and the title-length are ***independent***. Indents do not apply to titles; page-offsets do.
Notes:	E, **m**

15.8. Input and Output when using troff

We now describe two *nroff* requests that we omitted earlier, because their usefulness is more apparent when you understand the *nroff* command line. Normally *nroff* takes its input from the files given when it is called up. However there are ways in which the formatter can be made to take part of its input from elsewhere, using *nroff* requests embedded in the document text.

15.8.1. **.so** — *Read Text from a File*

The **.so** request, which tells *nroff* to **s**witch **o**ver and take its **so**urce from the named file. For example, suppose you have a set of macros that you have defined, and you have them in a file called 'macros'. We can call them up from the *nroff* command line:

```
tutorial% nroff macros document
tutorial%
```

as we showed earlier, but it's a bit of a nuisance having to do this all the time. Also, if only some of our documents use the macros, and others don't, it can be difficult to remember which is which. An alternative is to make the first line of the 'document' file look like this:

```
.so macros
```

Now we can format the document by:

```
tutorial% nroff document
tutorial%
```

The first thing *nroff* sees in the file 'document' is the request **.so macros** which tells it to read input from the file called 'macros'. When it finishes taking input from 'macros', *nroff* continues to read the original file 'document'.

Another way of using the **.so** request lets you format a complete document, held in several files, by only giving one filename to the *nroff* command. Let us create a file called 'document' containing:

```
.so macros
.so section.1
.so section.2
.so section.3
            and so on through the document until . . .
.so appendix.C
```

We can now format it with the *nroff* command line:

```
tutorial% nroff document | lpr
tutorial%
```

This is a lot easier than typing all the filenames each time you format the document, and a lot less prone to error.

This technique is especially useful if your filenames reflect the contents of the various sections, rather than the order in which they appear. For instance, look at this file which describes a

whole book (something like the one you are reading):

```
tutorial% cat book
.so bookmacros
.so preface
.so intro
.so login      \"Getting Started on the UNIX System
.so directs    \"Directories and the File System
.so stdio      \"Commands, Processes, and Standard Files
          <etc...>
.so biblio     \"Bibliography
tutorial%
```

It is obviously much easier to format the whole thing with an *nroff* command line like this:

```
tutorial% nroff book | lpr
tutorial%
```

than it would be if you had to supply all the filenames in the right order. Notice that we used the comment feature of *nroff* to tie chapter titles to filenames.

Summary of the .so Request

Item	*Description*
Form of Request:	**.so** *filename*
Explanation:	Switch source file — The top input (file reading) level is switched to *filename*. The effect of an **so** in a macro is felt when **so** is encountered. When the new file ends, input is again taken from the original file. **so**'s may be nested.

Summary of the .nx Request

Item	*Description*
Form of Request:	**.nx** *filename*
If No Argument:	end-of-file
Explanation:	Next file is *filename*. The current file is considered ended, and the input is immediately switched to *filename*.

Summary of the .pi Request

Item	*Description*
Form of Request:	**.pi** *program_name*
Explanation:	Pipe output to *program* (*nroff* only). This request must occur *before* any printing occurs. No arguments are transmitted to *program*.

15.8.2. **.rd** — *Read from the Standard Input*

The input can be temporarily switched to the system ***standard input*** with **rd**, which will switch back when ***two*** newlines in a row are found (the ***extra*** blank line is not used). This mechanism is intended for insertions in form-letter-like documentation. On UNIX, the ***standard input*** can be the user's keyboard, a ***pipe***, or a ***file***.

Another *nroff* request that switches input from the file you specify is the **.rd** (read) request. The **rd** request reads an insertion from the standard input. When *nroff* encounters the **.rd** request, it prompts for input by sounding the terminal bell. A visible prompt can be given by adding an argument to **.rd**, as we show in the example below.

Everything typed up to a blank line (two newline characters in a row) is inserted into the text being formatted at that point. This can be used to 'personalize' form letters. If you have an input file with this text:

```
.po 10
.nf
.in 20
14th February
.in 0
Dear
.rd who
     Will you be my Valentine?
     If you will, give me a sign
     (I like roses, I like wine).
```

then when you format it, you will be prompted for input:

```
tutorial% nroff valentine | lpr
who:Peter

tutorial%
```

After typing the name Peter you have to press the RETURN key twice, since *nroff* needs a blank line to end input. The results of formatting that file is:

```
                                    14th February


          Dear Peter
               Will you be my Valentine?
               If you will, give me a sign
               (I like roses, I like wine).
```

To get another copy of this for Bill, you just run the *nroff* command again:

```
tutorial% nroff valentine | lpr
who:Bill

tutorial%
```

and again for Joe, and for Manuel, and Louis, and Alphonse, and ...

Since ***nroff*** takes input from the terminal up to a blank line, you are not limited to a single word, or even a single line of input. You can use this method to insert addresses or anything else into form letters.

Summary of the .ex Request

Item	*Description*
Form of Request:	**.ex** ***prompt***
Explanation:	Exit from ***nroff/troff***. Text processing is terminated exactly as if all input had ended.

Summary of the .rd Request

Item	*Description*
Form of Request:	**.rd** ***prompt***
Initial Value:	Not applicable
If No Argument:	***prompt***=BEL
Explanation:	Read insertion from the standard input until two newlines in a row are found. If the standard input is the user's keyboard, ***prompt*** (or a BEL) is written onto the user's terminal. **rd** behaves like a macro, and arguments may be placed after ***prompt***.

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*** simultaneously come 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** (§19); the process would ultimately be ended by an **ex** in the insertion file.

15.8.3. **.tm** — *Send Messages to the Standard Error File*

The **.tm** (terminal **m**essage) request displays a message on the standard error file. The request looks like:

```
.tm tell me some good news
```

and when ***troff*** or ***nroff*** encounters this in the input file, it displays the string

```
tell me some good news
```

on the standard ***error*** file. This request has been used in older versions of the **−ms** macro package to rebuke the user when (for instance) an abstract for a paper was bigger than a page. Other macro packages use the **.tm** request for assisting in generating tables of contents and indices and such supplementary material.

Summary of the .tm Request

Item	*Description*
Form of Request:	**.tm** ***string***
Initial Value:	Not applicable
If No Argument:	Display a newline
Explanation:	After skipping initial blanks, ***string*** (rest of the line) is read in ***copy mode*** and written on the user's terminal.

15.9. Using Strings as Shorthand

Obviously if a paper contains a large number of occurrences of an acute accent over a letter 'e', typing **\o"e\'"** for each é would be a great nuisance.

Fortunately, ***troff*** provides a way in which you can 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 few editing changes. A reference to a string is replaced by whatever text the string was defined as.

A ***string*** is a named string of ***characters***, ***not*** including a newline character, that may be interpolated by name at any point in your text. Note that names of ***troff*** requests, names of macros, and names of strings, all share the ***same*** name list. String names may be one or two characters long and may usurp previously defined request, macro, or string names.

You create a string (and give it an initial value) with the **.ds** (**d**efine **s**tring) request. You can later add more characters to the end of the string by using the **.as** (**a**ppend to **s**tring) request.

You get the value of a string placed in the text, where it is said to be ***interpolated***, by using the notation:

***x**

for a one-character string name called ***x***, and the more complicated notation:

***(xx**

for a two-character string name ***xx***. String references and macro invocations may be nested.

Strings (just like macros) can be renamed with the **.rn** (**r**e**n**ame) request, or can be removed from the namelist with the **.rm** (**r**e**m**ove) request.

15.9.1. **.ds** — *Define Strings*

You create a string (and define its initial value) with the **.ds** (**d**efine **s**tring) request. 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 referred to 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.

As another live example, in the section on ***Ligatures*** we noted that ***troff*** doesn't know about the Scandinavian ligatures — you have to make them up for yourself. Here are the definitions of the strings for those ligatures:

```
.ds ae a\h'-(\w'a'u*4/10)'e
.ds Ae A\h'-(\w'A'u*4/10)'E
.ds oe o\h'-(\w'o'u*4/10)'e
.ds Oe O\h'-(\w'O'u*4/10)'E
```

See the section on ***Arbitrary Horizontal Motions with the \h Function*** for a discussion on what the **\h** constructs are doing in the string definitions above. Having defined the strings, all you have to do is type the string references like this:

```
... the Scandinavian ligatures \*(oe, \*(ae, \*(Oe, and \*(Ae ...
```

in order to get ... the Scandinavian ligatures œ, æ, Œ, and Æ ... into your stream of text.

If a string must begin with spaces, define it as

```
.ds xx "      text
```

The double quote signals the beginning of the definition. There is no trailing quote — 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 is thrown away and the next line added 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.

Summary of the .ds Request

Item	*Description*
Form of Request:	**.ds** ***xx string***
Initial Value:	Not applicable
If No Argument:	Ignored
Explanation:	Define a string ***xx*** containing ***string***. Any initial double-quote in ***string*** is stripped off to permit initial spaces.

15.9.2. **.as** — *Append to a String*

The **.as** (**append** to **s**tring) request adds characters to the end of a string. You use the **.as** request like this

```
.as xx string-of-characters
```

where ***string-of-characters*** is appended to the end of whatever is already in the string ***xx***.

Note that the string mentioned in a **.as** request is ***created*** if it didn't already exist, so in that respect an initial **.as** request acts just like a **.ds** request.

For example, here's a short fragment from the **.H** macro that was used to generate the section numbers in this document. The **.H** macro is called up like

```
.H level-number "Text of the Title"
```

where ***level-number*** is 1, 2, 3, ... to indicate that this is a first, second, third, ... level heading. The **.H** macro keeps track of the various section numbers via a bunch of number registers H1 through H5, and they are tested for and appended to the **SN** string if appropriate. For example:

```
.ds SN \\n(H1.                      set the initial section number string
.if \\n(NS>1 .as SN \\n(H2.         append H2 if needed
.if \\n(NS>2 .as SN \\n(H3.         append H3 if needed
.if \\n(NS>3 .as SN \\n(H4.         append H4 if needed
.if \\n(NS>4 .as SN \\n(H5.         append H5 if needed
                    .
                    .
          more processing to compute indentations and such ...
                    .
                    .
\\*(SN\\ \\ \t\c                    Now output the text
\&\\$2
                    .
                    .
          and yet more processing ...
                    .
                    .
```

Let's unscramble that mess. The essential parts are the initial line that says:

```
.ds SN \\n(H1.                      set the initial section number string
```

which sets the *SN* (**S**ection **N**umber) string to the value of the H1 number register that counts chapter level numbers. Then the following four lines essentially all perform a test that say:

> **.if** the ***level-number*** is greater than *N*, append the next higher section counter to the string. That is, if the current section number is greater than 2, we append the value of the level 3 counter, then if the section number is greater than 3, we append the value of the level 4 counter, and so on.

Finally, the built-up *SN* string, followed by the text of the title, gets placed into the output text with the lines that read:

```
\\*(SN\\ \\ \t\c                    Now output the text
\&\\$2
```

And in fact we can use the mechanisms that exist to play games like that because we are using a macro package to format ***this*** document, and those number registers are available to us. So we can define a string like this:

```
.ds XX \n(H1-
```

and interpolate that string like this:

```
\*(XX
```

to get the value

```
15-
```

printed in the text. Now we can ***append*** the rest of the section counters to that *XX* string like this (without caring whether they have any values):

```
.as XX \n(H2-\n(H3-\n(H4-\n(H5
```

and *then* when we interpolate that string we get this:

15-9-2-0-0

which if you look, should be the section number of the stuff you are now reading.

Summary of the .ds Request

Item	*Description*
Form of Request:	**.ds** ***xx string***
Initial Value:	Not applicable
If No Argument:	Ignored
Explanation:	Append ***string*** to string ***xx*** (append version of **ds**). The string ***xx*** is created if it didn't already exist.

15.10. Macros, Diversions, and Traps

15.10.1. Macros

Before we can go much further in ***troff*** or ***troff***, we need to learn a bit about the macro facility. In its simplest form, a ***macro*** is just a shorthand notation quite similar to a string. But, whereas a string is somewhat limited, a macro can not only contain multiple lines of text and requests, but a macro can also deal with arguments that can change its behavior from one invocation to the next.

A ***macro*** is a named set of arbitrary ***lines*** that may be invoked by name or with a ***trap***. Request, macro, and string names share the ***same*** name list. Macro 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. A macro is invoked in the same way as a request; a control line beginning ***.xx*** interpolates the contents of macro ***xx***. The remainder of the line may contain up to nine ***arguments***. String references and macro invocations may be nested.

15.10.1.1. **.de** — *Define a Macro*

Suppose we want every paragraph to start in exactly the same way — with a space and a temporary indent of two ems. We show a (very simplified) version of the **.PP** (paragraph) macro from the **−ms** macro package:

```
.sp
.ti +2m"
```

Then to save typing, we would like to collapse these into one shorthand line, a ***troff*** 'request' like

```
.PP
```

that would be treated by ***troff*** exactly as if you had typed:

```
.sp
.ti +2m
```

.PP is called a ***macro***. The way we tell ***troff*** what **.PP** means is to ***define*** it with the **.de** (**de**fine) request:

```
.de PP
.sp
.ti +2m
..
```

The first line names the macro (we used **.PP** for which is a standard notation for 'paragraph'. It is common practice to use upper-case names for macros so that their names don't conflict with ordinary ***troff*** requests. The last line **..** marks the end of the definition. In between is the text (often called the ***replacement text***), which is simply inserted whenever ***troff*** sees the 'request' or

macro call

```
.PP
```

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 requests is critically important. Not only does it save typing, but it makes later changes much easier. Suppose we 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, we 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 we used **.PP**.

\" is an in-line *troff* function that means that the rest of the line is to be ignored. We use it here to add comments to the macro definition (a wise idea once definitions get complicated).

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 requests **.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.

If later on we decide that the indent should be 0.5i, then it is only necessary to change the definitions of **.BS** and **.BE**, not the whole paper.

15.10.1.2. Macros with Arguments

The next step is 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, we need two things: first, when we define the macro, we have to indicate that some parts of it will be provided as arguments when the macro is called. Then when the macro is called we have to provide actual arguments to be plugged into the definition.

When a macro is invoked by name, the remainder of the line is taken to contain up to nine arguments. The argument separator is the space character, and arguments may be surrounded by double-quotes to permit embedded space characters. Pairs of double-quotes may be embedded 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 preceding \. The number of currently available arguments is in the .$ register.

No arguments are available at the top (non-macro) level in this implementation. 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.

Summary of the .de Request

Item	*Description*
Form of Request:	**.de** *xx yy*
Initial Value:	Not applicable
If No Argument:	*.yy*=..
Explanation:	Define or redefine the macro *xx*. 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 *.yy*, whereupon the macro *yy* is called. In the absence of *yy*, the definition is terminated by a line beginning with '..'. A macro may contain **de** requests provided the terminating macros differ or the contained definition terminator is concealed. '..' can be concealed as \\.. which will copy as \.. and be reread as '..'.

15.10.1.3. **.am** — *Append to a Macro*

Summary of the .am Request

Item	*Description*
Form of Request:	**.am** *xx yy*
Initial Value:	Not applicable
If No Argument:	*.yy*=..
Explanation:	Append to macro (append version of **de**).

15.10.1.4. **.rm** — *Remove Requests, Macros, or Strings*

Summary of the .rm Request

Item	*Description*
Form of Request:	**.rm** *F*
Initial Value:	Not applicable
If No Argument:	Ignored
Explanation:	Remove request, macro, or string. The name *xx* is removed from the name list and any related storage space is freed. Subsequent references will have no effect.

15.10.1.5. **.rn** — *Rename Requests, Macros or Strings*

Summary of the .rn Request	
Item	*Description*
Form of Request:	**.rn** *xx yy*
Initial Value:	Not applicable
If No Argument:	Ignored
Explanation:	Rename request, macro, or string *xx* to *yy*. If *yy* exists, it is first removed.

Let us illustrate by defining a macro **.SM** that will print its argument two point sizes 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.

By the way, 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
..
```

The **\h** and **\w** commands need no extra backslash, as we discussed above. 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 produces section headings rather like those in this paper, with the sections numbered automatically, and the title in bold in a smaller size. The use is

```
.SH  "Section title ..."
```

If the argument to a macro is to contain spaces, then 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. A number register may have the same name as a macro without conflict but a string may not.

We used **\\n(SH** instead of **\n(SH** and **\\n(PS** instead of **\n(PS**. If we had used **\n(SH**, we would get the value of the register at the time the macro was ***defined***, not at the time it was ***used***. If that's what you want, fine, but not here. 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, but a user could supply private definitions for any of the strings.

15.10.1.6. Copy Mode Input Interpretation

During definition and extension of strings and macros (not by diversion) the input is read in ***copy mode***. The input is copied without interpretation ***except*** that:

- 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 (§9).
- \\ 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.

15.10.2. Using Diversions to Store Text for Later Processing

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. A diversion is very similar to a macro and in fact uses the same mechanisms as the macro facility. Any part of the output may be sent into a diversion instead of being printed, and then at some convenient time the diversion may be brought back into the input.

15.10.2.1. **.di** *— Divert Text*

The request **.di xy** begins a diversion — all subsequent output is collected into the diversion called **xy** until a **.di** request with no argument is encountered, which terminates the diversion. The processed text is available at any time thereafter, simply by giving the request

```
.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 requests **.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** is 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 write it in full generality. This section is not intended to teach everything about diversions, but to sketch out enough that you can read existing macro packages with some comprehension.

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 embed in the diversion the appropriate **cs** or **bd** requests with the *transparent* mechanism described in the ***Introduction to nroff and troff***.

Diversions may be nested and certain parameters and registers are associated with the current diversion level (the top non-diversion level may be thought of as the 0th diversion level). These are the diversion trap and associated macro, no-space mode, the internally-saved marked place (see **mk** and **rt**), the current vertical place (**.d** register), the current high-water text base-line (**.h** register), and the current diversion name (**.z** register).

Summary of the .di Request	
Item	*Description*
Form of Request:	**.di** *xx*
Initial Value:	Not applicable
If No Argument:	End of diversion
Explanation:	Divert output to macro *xx*. 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.
Notes:	D

15.10.2.2. **.da** — *Append to a Diversion*

Summary of the .da Request	
Item	*Description*
Form of Request:	**.da** *xx*
Initial Value:	Not applicable
If No Argument:	End of diversion
Explanation:	Append to diversion *xx*. This is the diversion equivalent of the **.am** (append to macro) request.

15.10.3. Using Traps to Process Text at Specific Places on a Page

Three types of trap mechanisms are available, namely ***page traps***, ***diversion traps***, and ***input-line-count traps***.

Macro-invocation traps may be planted using the **.wh** (**wh**en) request at any page position including the top. This trap position may be changed using the **.ch** (**ch**ange) request. Trap positions at or below the bottom of the page have no effect unless or until 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 ***sweeps past*** 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 is the distance to

the page bottom.

A macro-invocation trap effective in the current diversion may be planted using the **.dt** (diversion trap) request. 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 the **.it** request below.

15.10.3.1. **.wh** — *Set Page or Position Traps*

Summary of the .wh Request

Item	*Description*
Form of Request:	**.wh** *N xx*
Initial Value:	Not applicable
If No Argument:	Not applicable
Explanation:	Install a trap to invoke *xx* at page position *N;* a ***negative*** *N* will be interpreted with respect to the page ***bottom***. Any macro previously planted at *N* is replaced by *xx*. A zero *N* refers to the ***top*** of a page. In the absence of *xx*, the first found trap at *N*, if any, is removed.
Notes:	**v**

15.10.3.2. **.ch** — *Change Position of a Page Trap*

Summary of the .ch Request

Item	*Description*
Form of Request:	**.ch** *xx N*
Initial Value:	Not applicable
If No Argument:	Not applicable
Explanation:	Change the trap position for macro *xx* to be *N*. In the absence of *N*, the trap, if any, is removed.
Notes:	**v**

15.10.3.3. **.dt** — *Set a Diversion Trap*

Summary of the .dt Request

Item	*Description*
Form of Request:	**.dt** *N zz*
Initial Value:	Not applicable
If No Argument:	Turn off diversion trap
Explanation:	Install a diversion trap at position *N* in the ***current*** diversion to invoke macro ***zz***. Another **dt** will redefine the diversion trap. If no arguments are given, the diversion trap is removed.
Notes:	D, **v**

15.10.3.4. .it — Set an Input-Line Count Trap

Summary of the .it Request

Item	*Description*
Form of Request:	**.it** *N zz*
Initial Value:	Not applicable
If No Argument:	Turn off trap
Explanation:	Set an input-line-count trap to invoke the macro ***zz*** after *N* lines of ***text*** 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.
Notes:	E

15.10.3.5. .em — Set the End of Processing Trap

Summary of the .em Request

Item	*Description*
Form of Request:	**.em** ***zz***
Initial Value:	Not applicable
If No Argument:	No trap installed
Explanation:	Call the macro ***zz*** when all input has ended. The effect is the same as if the contents of ***zz*** had been at the end of the last file processed.

15.11. Number Registers and Arithmetic

In a programmable text formatter such as *troff*, you need a facility for storing numbers somewhere, retrieving the numbers, and for doing arithmetic on those numbers. *troff* meets this need by providing things called *number registers*. Number registers give you the ability to define variables where you can place numbers, retrieve the values of the variables, and do arithmetic on those values. Number registers, like strings and macros, can be useful in setting up a document so it is easy to change later. And of course number registers serve for any sort of arithmetic computation.

Number registers, just like strings, have one- or two-character names. They are set by the **.nr** (**n**umber **r**egister) request, and are referenced anywhere by **\n***x* (one character name) or **\n(***xy* (two character name). When you access a number register so that its value appears in the printed text, the jargon says that you have *interpolated* the value of the number register.

A variety of parameters are available to the user as predefined, named *number registers* (see Appendix B, the *troff* Request Summary, and Appendix D, Predefined Number Registers). In addition, the user may define his own named registers. Register names are one or two characters long and *do not* conflict with request, macro, or string names. Except for certain predefined 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*.

troff defines several pre-defined number registers (see Appendix D). Among them are **%** 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 number registers can be used in computations like any other register, but some, like **.s** and **.f**, cannot be changed with an **.nr** request because they are *read only*.

15.11.1. **.nr** — *Set Number Registers*

You create and modify number registers using the **.nr** (**n**umber **r**egister) request. In its simplest form, the **.nr** request just places a value into a number register — the register is created if it doesn't already exist. The **.nr** request specifies the *name* of the number register, and also specifies the *initial value* to be placed in there. So the request

```
.nr PD 1.5v
```

would be a request to set a register called **PD** (which we might know as 'Paragraph Depth' if we were writing a macro package) to the value 1.5v (1.5 of *troff*'s vertical units).

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 (see the Chapter *Formatting Documents with the '−ms' Macro Package*). 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, a user may say:

```
.nr PS 9
.nr VS 11
```

The 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 **PS** and **VS** number registers.

Why are there two backslashes? When *troff* originally reads the macro definition, it peels off one backslash to see what's coming next. 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, point size and vertical spacing will be frozen at the time the macro is *defined*, not when the macro is *used*.

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.

Summary of the .nr Request

Item	*Description*
Form of Request:	**.nr** *R* ±*N M*
Initial Value:	Not applicable
If No Argument:	Ignored
Explanation:	Assign the value ±*N* to number register *R*, with respect to the previous value, if any. Set the increment for auto-incrementing to *M*.
Notes:	**u**

15.11.1.1. Auto-increment Number Registers

When you set a number register with the **.nr** request, you can also specify an additional number as an auto-increment value — that is, the number is added to the number register every time you access the number register. You specify the auto-increment value with a request such as:

```
.nr sn 0 1
```

to specify a (hypothetical) **s**ection **n**umber register that starts off with the value 0 and is incremented by 1 every time you use it. This might find application (for instance) in numbering the sections of a document automatically — something you might expect a computer to do for you, no? You might also define a numbered list macro that would clock up the item number every time you started a new item.

Here's a very quick and dirty example of the use of auto-incrementing a number register:

```
.nr cn -1 2
...
the odd numbers \n+(cn, \n+(cn, \n+(cn, \n+(cn, \n+(cn, \n+(cn,
...
```

When we format the above sequence, we get the following:

... the odd numbers 1, 3, 5, 7, 9, 11, ...

The table below shows the effects of accessing the number registers *x* and *xx* after a **.nr** request that sets them to the value *N* with an auto-increment value of *M*.

Table 15-9: Access Sequences for Auto-incrementing Number Registers

Request	*Access Sequence*	*Effect on Register*
.nr N M	**\n***x*	none
.nr N M	**\n(***xx*	none
.nr N M	**\n+***x*	*x* incremented by *M*
.nr N M	**\n−***x*	*x* decremented by *M*
.nr N M	**\n+(***xx*	*xx* incremented by *M*
.nr N M	**\n−(***xx*	*xx* decremented by *M*

15.11.2. Arithmetic Expressions with Number Registers

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 and logical operators as shown in the table below. Parts of an expression can be surrounded by parentheses.

Table 15-10: Arithmetic Operators and Logical Operators for Expressions

Arithmetic Operator	*Meaning*
+	Addition
−	Subtraction
/	Division
*	Multiplication
%	Modulo
Logical Operator	***Meaning***
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
= **or** ==	Equal to
&	and
:	or

Except where controlled by parentheses, evaluation of expressions is left-to-right — there is no operator precedence.

Although the arithmetic we have done so far has been straightforward, more complicated things are somewhat tricky. First, number registers hold only integers. ***troff*** arithmetic uses truncating integer division. Second, in the absence of parentheses, evaluation is done from left to right without any operator precedence (including relational operators). Thus

```
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 where you would not 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 obvious enough — 3.5 inches. Sorry — remember that the default units for horizontal parameters like the **.ll** request are ems. So that expression is really '7 ems / 2 inches', and when translated into machine units, it becomes zero. How about

```
.ll 7i/2
```

Still no good — the '2' is '2 ems', so '7i/2' is small, although not zero. You ***must*** use

```
.ll 7i/2u
```

So again, a safe rule is to attach a scale indicator to every number, even constants.

For arithmetic done within a **.nr** request, 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** request.

15.11.3. **.af** — *Specify Format of Number Registers*

When you use a number register as part of the text, the contents of the register are said to be *interpolated* into the text at that point. For example, you could use the following sequence:

```
.nr  xy  567
...
the value of the \fIxy\fP number register is: \n(xy.
...
```

and when you formatted that sequence, it would appear as:

... the value of the ***xy*** number register is: 567. ...

When interpolated, the value of the number register is read out as a decimal number. You can change this format by using the **.af** (**a**ssign **f**ormat) request to get things like Roman numerals or sequences of letters. Here is the example of the auto-incrementing section above, but with the interpolation format now set for lower-case Roman numerals:

```
.nr cn -1 2
.af cn i
...
the odd Roman numerals \n+(cn, \n+(cn, \n+(cn, \n+(cn, \n+(cn, \n+(cn,
...
```

When we format the above sequence, we get the following:

... the odd Roman numerals i, iii, v, vii, ix, xi, ...

A decimal format having *N* digits specifies a field width of *N* digits.

Read-only number registers and the ***width*** function are always decimal.

The table below shows the different formats you can apply to a number register when it is interpolated.

Table 15-11: Interpolation Formats for Number Registers

Format	*Description*	*Numbering Sequence*
1	Decimal	0, 1, 2, 3, 4, 5, ...
001	Decimal with leading zeros	000, 001, 002, 003, 004, 005, ...
i	Lower-case Roman Numerals	0, i, ii, iii, iv, v, ...
I	Upper-case Roman Numerals	0, I, II, III, IV, V, ...
a	Lower-case Letters	0, a, b, c, ..., z, aa, ab, ..., zz, aaa, ...
A	Upper-case Letters	0, A, B, C, ..., Z, AA, AB, ..., ZZ, AAA, ...

Summary of the .af Request	
Item	*Description*
Form of Request:	**.af** *R c*
Initial Value:	Arabic
If No Argument:	Ignored
Explanation:	Assign format *c* to register *R*.

15.11.4. **.rr** — *Remove Number Registers*

If you create many number registers dynamically, you may well have to remove number registers that you aren't using any more to recapture internal storage space for newer registers. You remove a number register with the **.rr** (**r**emove **r**egister) request:

```
.rr xy
```

removes the *xy* number register from the list.

Summary of the .rr Request	
Item	*Description*
Form of Request:	**.rr** *R*
Initial Value:	Not applicable
If No Argument:	Ignored
Explanation:	Remove register *R*. If many registers are being created dynamically, it may become necessary to remove no longer used registers to recapture internal storage space for newer registers.

15.12. Arbitrary Motions and Drawing Lines and Characters

This section is a grab-bag of functions for moving to arbitrary places on the page and for drawing things. This section covers a number of useful topics:

- Local motions — how to move forward and backward and up and down on the page to get special effects.
- Constructing whole characters out of pieces of characters that are available in the special font — these facilities are for doing mathematical typesetting.
- Drawing horizontal and vertical lines to make boxes and underlines and such.
- Various types of padding characters, zero-width characters, and functions for obtaining the width of a character string.

Most of these commands are straightforward, but messy to read and tough to type correctly.

15.12.1. **\u** *and* **\d** *Functions — Half-Line Vertical Movements*

If you won't use *eqn*, subscripts and superscripts are most easily done with the half-line local motions **\u** (for up) and **\d** (for down). To go ***up*** the page half a point-size, insert a **\u** at the desired place, and to go ***down*** the page half a point-size, insert a **\d** at the desired place. The **\u** and **\d** in-line functions should always be used in pairs, as explained below. Thus if your input consists of the following fragment:

```
... area of a circle is `Area = \(*pr\u2\d' when calculating ...
```

the output when that fragment is formatted consists of:

... area of a circle is 'Area = πr^{2}' when calculating ...

This is a first approximation of what you want, but the superscript '2' is too large. To make the '2' smaller, bracket it with **\s-2...\s0**. This reduces the point-size by two points before the superscript and raises the point-size by two points after the superscript. This example input:

```
... area of a circle is `Area = \(*pr\u\s-2s\s0\d' when calculating ...
```

when formatted, generates:

... area of a circle is 'Area = πr^{2}' when calculating ...

Now the reason that the **\u** and **\d** functions should always be correctly paired is that they refer to the ***current*** point size, so you must be sure to put any local motions either both inside or both outside any size changes, or you will get an unbalanced vertical motion. Carrying this example further, the input could look like this:

```
... area of a circle is `Area = \(*pr\u\s-22\d\s0' when calculating ...
```

We'll format that example in a larger point-size so that you can see the effect of the baseline being out of whack. So when we format the above construct with the motions incorrectly paired, we get this:

... area of a circle is 'Area = πr^2' when calculating

As you can see, the baseline is higher after the incorrectly-displayed equation.

15.12.2. Arbitrary Local Horizontal and Vertical Motions

The next two sections describe the in-line **\v** (vertical) and the **\h** (horizontal) local mtion functions.

The general form of these functions is **\v'***N***'** for the vertical motion function, and **\h'***N***'** for the horizontal motion function. The argument *N* in the functions is the distance to move. 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, it is necessary that the ***net*** vertical local motion within a word in filled text and otherwise within a line balance to zero.

15.12.2.1. \v Function — Arbitrary Vertical Motion

Sometimes the space given by **\u** and **\d** is not the right amount (usually too much). The in-line **\v** function requests an arbitrary amount of vertical motion. The in-line **\v** function

\v'*amount***'**

moves up or down the page by the amount specified in ***amount***. For example, here's how to get a large letter at the start of a verse:

```
.in +.3i
.ti -.3i
\v'1.0'\s36A\s0\v'-1.0'\h'-4p'wake! for Morning in the Bowl of Night
\h'2p'Has flung the Stone that puts the Stars to Flight:
.in -.3i
And Lo! the Hunter of the East has caught
The Sultan's Turret in a Noose of Light.
```

and when we format that verse we get:

Awake! for Morning in the Bowl of Night
Has flung the Stone that puts the Stars to Flight:
And Lo! the Hunter of the East has caught
The Sultan's Turret in a Noose of Light.[3]

The indent amount we used here (0.3 inch) was determined by fiddling around until it looked reasonable. Later we show another in-line function for measuring the actual width of something.

A minus sign means upward motion, while no sign or a plus sign means move down the page. Thus **\v'-1'** means an upward vertical motion of one line space.

---

[3] Omar Khayyám — *the Rubáiyát*

There are many other ways to specify the amount of motion. The following three examples are all legal.

```
\v'0.1i'
\v'3p'
\v'-0.5m'
```

Notice that the scale specifier (i, p, or m) goes inside the quotes. Any character can be used in place of the quotes; this is also true of all other *troff* commands 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.

15.12.2.2. **\h** *Function — Arbitrary Horizontal Motion*

Arbitrary horizontal motions are also available — **\h** is quite analogous to **\v**, except that the default scale factor is ems instead of line spaces. As an example,

```
\h'-0.1i'
```

causes a backwards motion of a tenth of an inch. As a practical matter, consider printing the mathematical symbol '>>'. The standard 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'*thing***'**

is a number equal to the width of 'thing' 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** for machine units, or the motion produced will be far too large. *troff* is quite happy with the nested quotes, by the way, so long as you don't leave any out.

As a live example of this kind of construction, the œ, æ, Œ, and Æ ligatures discussed in the section on *Ligatures* were constructed using the **\h** function to define the following strings:

```
.ds ae a\h'-(\w'a'u*4/10)'e
.ds Ae A\h'-(\w'A'u*4/10)'E
.ds oe o\h'-(\w'o'u*4/10)'e
.ds Oe O\h'-(\w'O'u*4/10)'E
```

and for any given one of those strings, the mess is unscrambled like this:

Construct	*Explanation*
.ds ae	*Define a string called 'ae'.*
a	*Letter 'a' in the string.*
\h'-(\w'a'u*4/10)'	*Move backwards 0.4 of the width of the letter 'a'.*
e	*Letter 'e' in the string.*

15.12.3. **\0** *Function — Digit Sized Spaces*

The in-line **\0** function 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 line justification and filling. You could use the digit space to get numerical columns correctly lined up. For example, suppose you have this list of items:

```
.nf
.ta 5n
Step    Description
.sp 5p
1.      Unpack the handy dandy fuse blower.
2.      Inspect for obvious shipping defects.
        .
        .
        .
9.      Find a wall socket.
10.     Insert handy dandy fuse blower in wall socket.
11.     Push red button to blow all fuses.
.fi
```

When you format this list of operations, you get this result:

Step Description

1. Unpack the handy dandy fuse blower.
2. Inspect for obvious shipping defects.

.
.
.

9. Find a wall socket.
10. Insert handy dandy fuse blower in wall socket.
11. Push red button to blow all fuses.

As you can see, the numbers do not line up at the decimal point, but instead are lined up on the left. Placing a space character in front of the digits in the input is not sufficient measure to line up the digits at the decimal. A space is not the same width as a digit (at least not in *troff*). A solution is to use the unpaddable digit-space character **\0** in front of the single digits like this:

```
.nf
.ta 5n
Step    \ODescription
.sp 5p
\01.    Unpack the handy dandy fuse blower.
\02.    Inspect for obvious shipping defects.
        .
        .
        .
\09.    Find a wall socket.
10.     Insert handy dandy fuse blower in wall socket.
11.     Push red button to blow all fuses.
.fi
```

Now when you format the text, you get this result:

Step Description

1. Unpack the handy dandy fuse blower.
2. Inspect for obvious shipping defects.

   .
   .
   .

9. Find a wall socket.
10. Insert handy dandy fuse blower in wall socket.
11. Push red button to blow all fuses.

which looks better than the previous example.

15.12.4. '\ ' Function — Unpaddable Space

There is also the in-line '\ ' function, which is the \ character followed by a space character. This function is an unpaddable character the width of a space. You can use this to make sure that things don't get split across line boundaries, for instance if you want to see something like **nroff −Tlp myfile** in the stream of text, with the command line set off like it was here and ensuring that it all apears on one line, you would type it in as `\ \ \fBnroff\ \-Tlp\ myfile\fP\ \` in-line in the text.

15.12.5. \| and \^ Functions — Thick and Thin Spaces

In typography, there are times when you need spaces that are one-sixth or one-twelfth of the width of an em-space. ***troff*** supplies the in-line \| function which is one-sixth of an em-space wide — this is sometimes called a 'thick space'. Where would you want such a thing? Well one place it ***could*** be used is in making an ellipsis look better. In general, an ellipsis in a proportional font looks too cramped if you just string three dots together:

...

and the dots tend to look too spread out if you just place spaces between them:

. . .

and so the answer is often to use the thick space to get a more pleasing effect like this:

. . .

which was actually achieved by typing:

```
.\|.\|.
```

Lastly, the in-line \^ function is one twelfth of the width of an em-space space. This function is almost always used for a typographical application called ***italic correction***. Consider an italic word followed by some punctuation such as *do tell*! Because the italic letters are slanted, they tend to lean slightly on the trailing punctuation, especially when the last letter is a tall one like the *l* in the example. So, what typographers do is to apply the ***italic correction*** in the form of a thin space just before the punctuation, so that the effect is now *do tell*! What we actually typed here was

```
\fIdo tell\fP\^!
```

with the italic correction just before the exclamation mark.

Typing the italic correction at every instance of adjacent Roman and Italic text, would be a lot of work. Some macro packages construct special-purpose macros for applying the italic correction. For example, the **−man** macro package has a **.IR** macro that joins alternating italic and Roman words together so that you can italicize parts of words or have italic text with trailing Roman punctuation. You use the **.IR** macro like:

```
.IR well spring
```

to get the composite effect of *well*spring in your text. The **.IR** macro (somewhat simplified) looks like this:

```
.de IR
\&\fI\\$1\^\fR\\$2\fI\\$3\^\fR\\$4\fI\\$5\^\fR\\$6\fI\\$7\^\fR\\$8\fI\\$9\^\fR
..
```

and you can see the italic correction applied after every parameter that is set in the italic font.

15.12.6. **\&** *Function — Non-Printing Zero-Width Character*

The **\&** function is a character that does not print, and does not take up any space in the output text. You might wonder what use it is at all? One application of the non-printing character used throughout this manual is to display examples of text containing ***troff*** or ***nroff*** requests. To print a ***troff*** request just as it appears in the input, you have to distinguish it from a real ***troff*** request. You cannot print an example whose input looks just like this:

```
.in +0.5i                    indent the text half an inch
        .
        .
        .
lots of lines of text to be processed
        .
        .
        .
.in -0.5i                    unindent the text half an inch
```

The . characters at the beginning of each line would be interpreted as *troff* requests instead of text representing examples of requests. In such cases, we have to use the **\&** function to stop *troff* or *nroff* from interpreting the . at the start of the line as a control character. We would type the example like this:

```
\&.in +0.5i                  indent the text half an inch
        \&.
        \&.
        \&.
lots of lines of text to be processed
        \&.
        \&.
        \&.
\&.in -0.5i                  unindent the text half an inch
```

Another place where the **\&** function is useful is within some of the other in-line functions such as the **\l** function. The **\l** function draws lines and you type the function like:

\l'*length character***'**

where ***length*** is the length of the line you want to draw, and ***character*** is the character to use. Sometimes, the ***character*** might look like a part of ***length***, for instance,

\l'1.0i='

doesn't get you a one-inch line of = signs as you might expect, because the = sign looks like an expression where you are trying to say that "1.0i is equal to" something else. When you encounter this situation, type the **\l** function like this:

\l'1.0i\&='

and the result is a one-inch line of ======== signs as you see.

15.12.7. **\o** *Function — Overstriking Characters*

Automatically-centered overstriking of up to nine characters is possible with the in-line **\o** (overstrike) function. The **\o** function looks like **\o'***string***'** where the characters in ***string*** are overprinted with their centers aligned. This means for example, that you can print from one to nine different characters superimposed upon each other. ***troff*** determines the width of this "character" you are creating to be the width of the widest character in your string. The superimposed characters are then centered on the widest character. The ***string*** should ***not*** contain local vertical motion.

The in-line **\o** function is used like this:

```
\o"set of characters"
```

This is useful for printing accents, as in

```
syst\o"e\(ga"me t\o"e\(aa"l\o"e\(aa"phonique
```

which produces

systѐme téléphonique

The accents are **\(ga** (grave accent) and **\(aa**, (acute accent) or **\`** and **\´**; remember that each is just one character to *troff*.

\o"e\´" produces é, and **\o"\(mo\(sl"** produces ∉.

15.12.8. **\z** *Function — Zero Motion Characters*

You can make your own overstrikes with another special convention, **\z**, the zero-motion command. **\z***x* 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:

is produced by

```
.sp 2
\s8\z\(ci\s14\z\(ci\s22\z\(ci\s36\z\(ci
```

The **.sp 2** line is needed to leave enough vertical space for the result.

As another 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
```

where '0.25m' is an empirical constant.

As further examples, **\z\(ci\(pl** produces

⊕

and **\(br\z\(rn\(ul\(br** produces the smallest possible constructed box:

There is also a more general overstriking function for piling things up vertically — this topic is discussed under ***Building Large Brackets with the \b Function***, later in this section.

15.12.9. **\w** *Function — Get Width of a String*

Back in the section on using tabs, we saw how we could set tab stops to various positions on the line and lay stuff out in columns based on the tab stops. Sometimes it is hard to figure out where the tab stops should go because you can't always tell in advance how wide things are — this is especially true for proportional fonts (by definition the characters aren't all the same size). Often what you want is to set tab stops based on the width of an item. Then you can set tab stops based on that width and remain independent of the size of the characters if you decide to change point size.

The in-line ***width*** function **\w'***string***'** generates the numerical width of ***string*** (in basic units). For example, **.ti -\w'1. 'u** could be used to temporarily indent leftward a distance equal to the size of the string '**1.** '. Size and font changes may be safely embedded in ***string***, and do not affect the current environment.

In a previous example we showed how a large capital letter could be placed in a verse with vertical motions and we played some games with indenting to get the thing to come out more-or-less right. The problem with that approach is that we had to measure the size of the character and arrive at the indent by trial and error (actually, error and trial). Another problem is that the measured indent didn't take the point-size into account — if we decide to change sizes, the measurements are all wrong. The width function can measure the size of the thing directly, so here's our example all over again using the **\w** function:

```
.in +\w'\s36A\s0'u
.ti -\w'\s36A\s0'u
\v'1.0'\s36A\s0\v'-1.0'\h'-5p'wake! for Morning in the Bowl of Night
\h'1p'Has flung the Stone that puts the Stars to Flight:
.in -\w'\s36A\s0'u
And Lo! the Hunter of the East has caught
The Sultan's Turret in a Noose of Light.
```

and when we format that text we get this result:

Awake! for Morning in the Bowl of Night
Has flung the Stone that puts the Stars to Flight:
And Lo! the Hunter of the East has caught
The Sultan's Turret in a Noose of Light.

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 **\n(stu-\n(sbu**. In ***troff*** the number register **ct** is set to a value between 0 and 3:

Number Register	Meaning
0	all of the characters in ***string*** were short lower case characters without descenders (like **e**)
1	at least one character has a descender (like **y**)
2	at least one character is tall (like **H**)
3	both tall characters and characters with descenders are present.

15.12.10. **\k** *Function — Mark Current Horizontal Place*

The in-line **\k***x* function stores the ***current*** horizontal position in the input line into register *x*. As an example, we could get a bold italic effect by the construction:

```
\kxword\h'|\nxu+2u'word
```

This emboldens ***word*** by backing up almost to its beginning and overprinting it, resulting in

word

15.12.11. **\b** *Function — Build Large Brackets*

The Special Mathematical Font contains a number of special characters for constructing large brackets out of pieces. The table below shows the escape-sequences for the individual pieces, what they look like, and their names.

Table 15-12: Pieces for Constructing Large Brackets

Escape Sequence	*Character*	*Description*
\(lt	⎧	left top of big curly bracket
\(lb	⎩	left bottom of big curly bracket
\(rt	⎫	right top of big curly bracket
\(rb	⎭	right bottom of big curly bracket
\(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 of big square bracket)
\(lc	⎡	left ceiling (left top of big square bracket)
\(rc	⎤	right ceiling (right top of big square bracket)

These pieces can be combined into various styles and sizes of brackets and braces by using the in-line **\b** (for **b**racketing) function. The **\b** function is used like this:

```
\b' string '
```

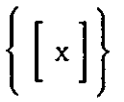
to pile up the characters vertically in ***string*** with the first character on top and the last on the bottom. The characters are vertically separated by one em and the total pile is centered 1/2 em above the current baseline (line in ***nroff***). For example:

```
\x'-0.5m'\x'0.5m'\b'\(lc\(lf'E\"\b'\(rc\(rf'
```

produces ⎡⎣E⎦⎤. As with previous examples, we should unscramble the whole mess for you:

Escape Sequence	*Character*	*Description*
\b		***start bracketing function***
\(lc	⎡	***left ceiling***
\(lf	⎣	***left floor***
E		***letter E***
\b		***start bracketing function***
\(rc	⎤	***right ceiling***
\(rf	⎦	***right floor***

Here's another example of using braces and brackets. You get this effect:



by typing this:

```
\b'\(lt\(lk\(lb' \b'\(lc\(lf' x \b'\(rc\(rf' \b'\(rt\(rk\(rb'
```

15.12.12. `\r` *Function — Reverse Vertical Motions*

The **\r** function makes a single reverse motion of one em upward in ***troff***, and one line upward in ***nroff***.

15.12.13. Drawing Horizontal and Vertical Lines

Typesetting systems commonly have commands to draw horizontal and vertical lines. Of course typographers don't call them lines — they are called 'rules' because once upon a time they were drawn with rulers. ***troff*** provides a convenient facility for drawing horizontal and vertical lines of arbitrary length with arbitrary characters, and these facilities are described in the subsections following.

15.12.13.1. `\l` *Function — Draw Horizontal Lines*

The in-line **\l** (lower-case ell) function draws a horizontal line. For example, the function **\l'1.0i'** draws a one-inch horizontal line like this _____ in the text.

The line is actually drawn using the ***baseline rule*** character in ***troff***, and the ***underline*** character in ***nroff***, but you can in fact make the character that draws the line any character you like by placing the character after the length designation. For example, you could draw a two inches of tildes by using **\l'2.0i~'** to get ~~~~~~ in the text. The construction **\L** is entirely analogous, except that it draws a vertical line instead of horizontal.

The general form of the **\l** function is

\l'*length character***'**

where ***length*** is the length of the string of characters to be drawn, and ***character*** is the character to use to draw the line. If ***character*** looks like a continuation of ***length***, you can insulate ***character*** from ***length*** with the zero-width **\&** sequence. If ***length*** is negative, a backward horizontal motion of size ***length*** is made ***before*** drawing the string. Any space resulting from ***length***/(size of ***character***) 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 ( ), the remainder space is covered by overlapping. If ***length*** is ***less*** than the width of ***character***, a single ***character*** is centered on a distance ***length***. As an example, here is a macro to underscore a string:

```
.de us
\\$1\l'~0\(ul'
..
```

and you use the **.us** macro like this:

```
.us "underlined words"
```

to yield underlined words in the stream of text. You could also write a macro to draw a box around a string:

```
.de bx
\(br\|\\$1\|\(br\l'|0\(rn'\l'|0\(ul'
..
```

and so you can type:

```
.bx "words in a box"
```

to get some words in a box in the text stream.

15.12.13.2. \L Function — Draw Vertical Lines

The in-line **\L** (upper-case ell) function draws a vertical line. As in the case of the **\l** function, the general form of the function is

\L'*length character*'

This draws a vertical line consisting consisting of the (optional) character ***character*** 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 base line. A positive ***length*** specifies a line drawn downward and a negative ***length*** specifies a line drawn upward. After the line is drawn ***no*** compensating motions are made; the instantaneous baseline is at the ***end*** of the line.

15.12.13.3. Combining the Horizontal and Vertical Line Drawing Functions

The horizontal and vertical line drawing functions may be used in combination to produce large boxes. The zero-width ***box-rule*** and the ½-em wide ***underrule*** were ***designed*** to form corners when using one-em vertical spacings. For example the macro

```
.de eb
.sp -1          \"compensate for next automatic base-line spacing
.nf             \"avoid possibly overflowing word buffer
\h'-.5n'\L'|\\nzu-1'\l'\\n(.lu+1n\(ul'\L'-|\\nzu+1'\l'|0u-.5n\(ul'
                \"draw box
.fi
..
```

draws a box around some text whose beginning vertical place was saved in number register *z* (using *.mk z*) as done for this paragraph.

15.12.14. **.mc** — *Place Characters in the Margin*

Many types of documents require placing specific characters in the margins. The most common use of this is placing bars down the margins to indicate what's changed in a document from one revision of a document to the next. This paragraph and the remainder of the text in this section were preceded by a

```
.mc \s10\(br\s0
```

request (that is, place a 10-point box-rule character in the margin) to turn on the marginal bars, and followed by a simple

```
.mc
```

request to turn off the marginal bars.

Summary of the .mc Request

Item	*Description*
Form of Request:	**.mc** *c N*
Initial Value:	Not applicable
If No Argument:	Turn off margin characters
Explanation:	Specifies that a ***margin*** character *c* appear a distance *N* 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 is appended to the line. If *N* is not given, the previous *N* is used; the initial *N* is 0.2 inches in ***nroff*** and 1 em in ***troff***. The margin character used with this paragraph was a 12-point box-rule.
Notes:	E, **m**

15.13. Input and Output Conventions and Character Translations

15.13.1. Input Character Translations

The newline delimits input lines. In addition, STX, ETX, ENQ, ACK, and BEL are accepted, and may be used as delimiters or translated into a graphic with a **.tr** (**tr**anslate) request (refer to the section on ***Input and Output Conventions and Character Translations***). All others are ignored.

15.13.2. **.ec** *and* **.eo** *— Set Escape Character or Stop Escapes*

The *escape* character \ introduces *escape sequences* — meaning the following character means another character, or indicates some function. A complete list of such sequences is given in a later chapter. The \ character should not be confused with the ASCII control character ESC of the same name. The escape character \ can be input with the sequence \\. The escape character can be changed with an **.ec** (**e**scape **c**haracter) request, and all that has been said about the default \ becomes true for the new escape character. **\e** can be used to print whatever the current escape character is. If necessary or convenient, the escape mechanism can be turned off with an **.eo** (**e**scape **o**ff) request and restored with the **.ec** request.

Summary of the .ec Request

Item	*Description*
Form of Request:	**.ec** *c*
Initial Value:	\
If No Argument:	\
Explanation:	Set escape character to \, or to *c*, if given.

Summary of the .eo Request

Item	*Description*
Form of Request:	**.eo**
Initial Value:	Escape mechanism is on
If No Argument:	Turn escape mechanism off
Explanation:	Turn escape mechanism off.

15.13.3. **.cc** *and* **.c2** *— Set Control Characters*

Both the control character . and the *no-break* control character ' may be changed, if desired. Such a change must be compatible with the design of any macros used in the span of the change, and particularly of any trap-invoked macros.

Summary of the .cc Request	
Item	***Description***
Form of Request:	**.cc** *c*
Initial Value:	.
If No Argument:	.
Explanation:	Set the basic control character to *c*, or reset to '.'.

Summary of the .c2 Request	
Item	***Description***
Form of Request:	**.c2** *c*
Initial Value:	'
If No Argument:	'
Explanation:	Set the ***no-break*** control character to *c*, or reset to ''.

15.13.4. **.tr** — *Output Translation*

One character can be made a stand-in for another character using the **.tr** (**tr**anslate) request. All text processing (for instance, character comparisons) takes place with the input (stand-in) character that appears to have the width of the final character. The graphic translation occurs at the moment of output (including diversion).

Summary of the .tr Request	
Item	***Description***
Form of Request:	**.tr** *abcd....*
Initial Value:	Not Applicable
If No Argument:	No translation
Explanation:	Translate *a* into *b*, *c* into *d*, etc. If an odd number of characters is given, the last one is mapped into the space character. To be consistent, a particular translation must stay in effect from ***input*** to ***output*** time.
Notes:	O

15.14. Automatic Line Numbering

15.14.1. **.nm** — *Number Output Lines*

Output lines may be numbered automatically via the **.nm** (**nu**mber) request. Refer to the following table for a summary of the **.nm** request. When in effect, a three-digit, arabic
3 number and a digit-space begins each line of output text. The text lines are thus offset by four digit-spaces, and otherwise retain their line length. To keep the right margin aligned with an earlier margin, you may want to reduce the line length by the equivalent of four
6 digit spaces. Blank lines, other vertical spaces, and lines generated by **tl** are ***not*** numbered. Numbering can be temporarily suspended with the **.nn** (**n**o **n**umber) request (see below), or with an **.nm** followed by a later **.nm +0**. In addition, a line number indent *I*, and the
9 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).

Summary of the .nm Request

Item	*Description*
Form of Request:	**.nm** ± *N M S*
Initial Value:	Line numbering turned off.
If No Argument:	Line numbering turned off.
Explanation:	Turn on line numbering if ±*N* is given. The next output line numbered is numbered ±*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**.
Notes:	E

15.14.2. **.nn** — *Stop Numbering Lines*

When you are using the **.nm** request to number lines (as discussed above), you can temporarily suspend the numbering with the **.nn** (**n**o **n**umber) request.

Summary of the .nn Request

Item	*Description*
Form of Request:	**.nn** *N*
Initial Value:	Not applicable
If No Argument:	*N*=1
Explanation:	The next *N* text output lines are not numbered.
Notes:	E

As an example, the paragraph portions of this section are numbered with *M*=3: **.nm 1 3** was placed at the beginning; **.nm** was placed at the end of the first paragraph; and **.nm +0** was placed in front of this paragraph; and **.nm** finally placed at the end. Line lengths were also changed (by **\w'0000'u**) to keep the right side aligned.

Another example is

```
.nm +5 5 x 3
```

which turns on numbering with the line number of the next line to be 5 greater than the last numbered line, *M*=5, spacing *S* is untouched, and with the indent *I* set to 3.

15.15. Conditional Processing of Input

Suppose we 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** request provides the 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, the rest of the line is treated as if it were text — here a request. If the condition is false, or zero or negative, the rest of the line is skipped.

It is possible to do more than one request 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 we will not go into here.

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.

In the following table, *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.

Summary of the .if Requests

Item	*Description*
Form of Request:	**.if** *c anything*
Initial Value:	Not Applicable
If No Argument:	Not Applicable
Explanation	If condition *c* true, accept *anything* as input. In multi-line case use \{*anything*\}.
Form of Request:	**.if !***c anything*
Explanation	If condition *c* false, accept *anything*.
Form of Request:	**.if** *N anything*
Explanation	If expression $N > 0$, accept *anything*.
Form of Request:	**.if !***N anything*
Explanation	If expression $N \leq 0$, accept *anything*.
Form of Request:	**.if** *'string1 'string2 ' anything*
Explanation	If *string1* identical to *string2*, accept *anything*.
Form of Request:	**.if !** *'string1 'string2 ' anything*
Explanation	If *string1* not identical to *string2*, accept *anything*.
Form of Request:	**.ie** *c anything*
Explanation	If portion of if-else (like above **if** forms).
Form of Request:	**.el** *anything*
Explanation	Else portion of if-else.

The built-in condition names are:

Table 15-13: Built In Condition Names for Conditional Processing

Condition Name	*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.

15.15.1. `.ig` — *Ignore Input Text*

Another mechanism for conditionally accepting input text is via the `.ig` (**ig**nore) request. Basically, you place the `.ig` request before a block of text you want to ignore:

```
.ig                 start of ignored block of text
                    .
                    .
                    .
block of text you don't want to appear in the printed output
                    .
                    .
                    .
..                  end of ignore block signalled with ..
```

The **.ig** request functions like a macro definition via the **.de** request except that the text between the .ig and the terminating .. is discarded instead of being processed for printing.

You can give the **.ig** request an argument — that is, an

```
.ig xy
```

request ignores all text up to and including a line that reads

```
.xy
```

which looks just like a request:

```
.ig ZZ              start of ignored block of text
                    .
                    .
                    .
block of text you don't want to appear in the printed output
                    .
                    .
                    .
.ZZ                         end of ignore block signalled with .ZZ
```

You can of course combine the .ig request with the other conditionals to ignore a block of text if a condition is satisfied. For example, you might want to omit blocks of text if the printed pages are destined for different audiences:

```
.nr W 1                     This manual is for Wizards only
                    .
                    .
                    .
          further processing
                    .
                    .
                    .
.if \nW .ig WZ              If the manual is for wizards
                    .
                    .
                    .
Tutorial material beneath the attention of wizards
                    .
                    .
                    .
.WZ                         end of ignored block of text
```

Summary of the .ig Request

Item	*Description*
Form of Request:	**.ig** *yy*
Initial Value:	Not applicable
If No Argument:	Ignore text up to a line starting with . .
Explanation:	Ignore input lines up to and including a line starting with .yy — use like the .de (define macro) request except that the input is discarded. The input is read in ***copy mode***, and any auto-incremented number registers will be affected.

15.16. Requests for Debugging your troff Input File

Sad to say, ***troff*** and ***nroff*** resemble languages for programming a typesetter rather than a mechanism to describe how a document should be put together. There are times when you just ***can't*** figure out why things are going wrong and not generating results as advertised. The requests described here are for dyed-in-the-wool macro wizards.

15.16.1. `.pm` — *Display Names and Sizes of Defined Macros*

The **.pm** (**p**rint **m**acros) request displays the names of all defined macros and how big they are. Why would ***anybody*** want to do such a thing? Well, if you're using a macro as a diversion, you might find out (by printing its size) that it is far bigger than you expect (like it's swallowing your entire file).

Summary of the .pm Request

Item	*Description*
Form of Request:	**.pm** *t*
Initial Value:	Not applicable
If No Argument:	All
Explanation:	Print macros. The names and sizes of all of the defined macros and strings are printed on the user's terminal; if *t* is given, only the total of the sizes is printed. The sizes is given in ***blocks*** of 128 characters.

15.16.2. `.fl` — *Flush Output Buffer*

The **.fl** (**fl**ush) request flushes the output buffer — this can be used when you're using ***nroff*** interactively.

Summary of the .fl Request

Item	*Description*
Form of Request:	**.fl**
Initial Value:	Not applicable
If No Argument:	Not applicable
Explanation:	Flush output buffer. Used in interactive debugging to force output.

15.16.3. **.ab** — *Abort*

A final useful request in the debugging category is the **.ab** (**ab**ort) request which basically bails out and stops the formatting.

Summary of the .ab Request

Item	*Description*
Form of Request:	**.ab** *text*
Initial Value:	Not applicable
If No Argument:	No text is displayed
Explanation:	Displays *text* and terminates without further processing. If *text* is missing, 'User Abort' is displayed. Does not cause a break. The output buffer is flushed.

15.17. Saving State with 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 six 'environments', each of which has independently settable versions of many of the parameters associated with processing, including size, font, line and title lengths, fill/nofill 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 in the range 0 thru 5. An **.ev** command 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.

When *troff* starts up, environment 0 is the default environment, so in general, the main text of your document is processed in this environment in the absence of any information to the contrary. Given this, we can modify the **.NP** (new page) macro 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 and is thus easier to understand and change.

Another major application for environments is for blocks of text that must be kept together.

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.

Summary of the .ev Request

Item	*Description*
Form of Request:	**.ev** *N*
Initial Value:	$N=0$
If No Argument:	Switch back to previous environment
Explanation:	Switch to environment N, where $0 \leq N \leq 5$. Switching is done in push-down fashion so that restoring a previous environment ***must*** be done with **.ev** rather than specific reference.

Appendix A

Examples of Fonts and Non–ASCII Characters

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 ¼em space. They are Times Roman, Italic, Bold, and a special mathematical font.

Times Roman

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + – . , / : ; = ? [ ] ¦
• □ — - _ ¼ ½ ¾ fi fl ff ffi ffl ° † ′ ¢ ® ©

Times Italic

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
*! $ % & ( ) ‘ ’ * + – . , / : ; = ? [ ] ¦*
• □ — - _ ¼ ½ ¾ fi fl ff ffi ffl ° † ′ ¢ ® ©

Times Bold

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + – . , / : ; = ? [ ] ¦
• □ — - _ ¼ ½ ¾ fi fl ff ffi ffl ° † ′ ¢ ® ©

Special Mathematical Font

" ´ \ ^ _ ` ~ / < > { } # @ + – = *
$\alpha\ \beta\ \gamma\ \delta\ \epsilon\ \zeta\ \eta\ \theta\ \iota\ \kappa\ \lambda\ \mu\ \nu\ \xi\ o\ \pi\ \rho\ \sigma\ \varsigma\ \tau\ \upsilon\ \phi\ \chi\ \psi\ \omega$
$\Gamma\ \Delta\ \Theta\ \Lambda\ \Xi\ \Pi\ \Sigma\ \Upsilon\ \Phi\ \Psi\ \Omega$
$\surd\ \geq\ \leq\ \equiv\ \sim\ \simeq\ \neq\ \rightarrow\ \leftarrow\ \uparrow\ \downarrow\ \times\ \div\ \pm\ \cup\ \cap\ \subset\ \supset\ \subseteq\ \supseteq\ \infty\ \partial$
§ ∇ ¬ ∫ ∝ ∅ ∈ ‡ ☞ ☜ | ○ ( )) { } | [ ] ⌈ ⌉

Non–ASCII characters and ***minus*** on the standard fonts.

Char	Input Name	Character Name
’	’	close quote
‘	`	open quote
—	\(em	3/4 Em dash
-	—	hyphen or
-	\(hy	hyphen
−	\−	current font minus
•	\(bu	bullet
□	\(sq	square
_	\(ru	rule
¼	\(14	1/4
½	\(12	1/2
¾	\(34	3/4

Char	Input Name	Character Name
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

Non-ASCII characters and ’, `, _, +, −, =, and * 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 followed by † which are mapped into upper case English letters in whatever font is mounted on font position one (default Times Roman). The special math plus, minus, and equals are provided to insulate the appearance of equations from the choice of standard fonts.

Char	Input Name	Character Name
+	\(pl	math plus
−	\(mi	math minus
=	\(eq	math equals
*	\(**	math star
§	\(sc	section
´	\(aa	acute accent
`	\(ga	grave accent
_	\(ul	underrule
/	\(sl	slash (matching backslash)
α	\(*a	alpha
β	\(*b	beta
γ	\(*g	gamma
δ	\(*d	delta
ϵ	\(*e	epsilon
ζ	\(*z	zeta
η	\(*y	eta
θ	\(*h	theta
ι	\(*i	iota
κ	\(*k	kappa
λ	\(*l	lambda
μ	\(*m	mu
ν	\(*n	nu
ξ	\(*c	xi
ο	\(*o	omicron
π	\(*p	pi

Char	Input Name	Character Name
ρ	\(*r	rho
σ	\(*s	sigma
ς	\(ts	terminal sigma
τ	\(*t	tau
υ	\(*u	upsilon
ϕ	\(*f	phi
χ	\(*x	chi
ψ	\(*q	psi
ω	\(*w	omega
	\(*A	Alpha†
	\(*B	Beta†
Γ	\(*G	Gamma
Δ	\(*D	Delta
	\(*E	Epsilon†
	\(*Z	Zeta†
	\(*Y	Eta†
Θ	\(*H	Theta
	\(*I	Iota†
	\(*K	Kappa†
Λ	\(*L	Lambda
	\(*M	Mu†
	\(*N	Nu†
Ξ	\(*C	Xi
	\(*O	Omicron†
Π	\(*P	Pi

Char	Input Name	Character Name
	\(*R	Rho†
Σ	\(*S	Sigma
	\(*T	Tau†
ϒ	\(*U	Upsilon
Φ	\(*F	Phi
	\(*X	Chi†
Ψ	\(*Q	Psi
Ω	\(*W	Omega
√	\(sr	square root
	\(rn	root en extender
≥	\(>=	>=
≤	\(<=	<=
≡	\(==	identically equal
≅	\(~=	approx =
≈	\(ap	approximates
≠	\(!=	not equal
→	\(->	right arrow
←	\(<-	left arrow
↑	\(ua	up arrow
↓	\(da	down arrow
×	\(mu	multiply
÷	\(di	divide
±	\(+-	plus-minus
∪	\(cu	cup (union)
∩	\(ca	cap (intersection)
⊂	\(sb	subset of
⊃	\(sp	superset of
⊆	\(ib	improper subset
⊇	\(ip	improper superset
∞	\(if	infinity
∂	\(pd	partial derivative
∇	\(gr	gradient
¬	\(no	not
∫	\(is	integral sign
∝	\(pt	proportional to
∅	\(es	empty set
∈	\(mo	member of
|	\(br	box vertical rule
‡	\(dd	double dagger
☞	\(rh	right hand
☜	\(lh	left hand
	\(bs	Bell System logo
|	\(or	or
○	\(ci	circle
⎧	\(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

Char	Input Name	Character Name
⌊	\(lf	left floor (left bottom of big square bracket)
⌋	\(rf	right floor (right bottom)
⌈	\(lc	left ceiling (left top)
⌉	\(rc	right ceiling (right top)

Appendix B

troff Request Summary

This appendix is a quick-reference summary of ***troff*** and ***nroff*** requests. In the following table, values separated by a **:** are for ***nroff*** and ***troff*** respectively.

The notes in column four are explained at the end of this summary.

Summary of troff and nroff Requests

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ab *text*	none	User Abort	—	Displays *text* and terminates without further processing; flush output buffer.
.ad *c*	adj,both	adjust	E	Adjust output lines with mode *c* from .j.
.af *R c*	arabic	—	—	Assign format to register *R* (*c*=**1**, **i**, **I**, **a**, **A**).
.am *xx yy*	—	*.yy*=..	—	Append to a macro.
.as *xx string*	—	ignored	—	Append *string* to string *xx*.
.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*.†
.bp ±*N*	N=1	—	B‡,v	Eject current page. Next page is number *N*.
.br	—	—	B	Break.
.c2 *c*	'	'	E	Set nobreak control character to *c*.
.cc *c*	.	.	E	Set control character to *c*.
.ce *N*	off	*N*=1	B,E	Center following *N* input text lines.
.ch *xx N*	—	—	v	Change trap location.

Summary of troff and nroff Requests

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.cs *F N M*	off	—	P	Constant character space (width) mode (font *F*).†
.cu *N*	off	*N*=1	E	Continuous underline in *nroff*; like ul in *troff*.
.da *xx*	—	end	D	Divert and append to *xx*.
.de *xx yy*	—	*.yy*=..	—	Define or redefine macro *xx*; end at call of *yy*.
.di *xx*	—	end	D	Divert output to macro *xx*.
.ds *xx string*	—	ignored	—	Define a string *xx* containing *string*.
.dt *N xx*	—	off	D,v	Set a diversion trap.
.ec *c*	\	\	—	Set escape character.
.el *anything*	—	—	—	Else portion of if-else.
.em *xx*	none	none	—	End macro is *xx*.
.eo	on	—	—	Turn off escape character mechanism.
.ev *N*	N=0	previous	—	Environment switched (*push down*).
.ex	—	—	—	Exit from *nroff/troff*.
.fc *a b*	off	off	—	Set field delimiter *a* and pad character *b*.
.fi	fill	—	B,E	Fill output lines.
.fl	—	—	B	Flush output buffer.
.fp *N F*	R,I,B,S	ignored	—	Font named *F* mounted on physical position $1 \leq N \leq 4$.
.ft *F*	Roman	previous	E	Change to font *F* = *x*, *xx*, or 1 through 4. Also \f*x*, \f(*xx*, \f*N*.
.fz *S F N*	none	—	—	Forces font *F* or *S* for special characters to be in size *N*.

Summary of troff and nroff Requests

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.hc *c*	\%	\%	E	Hyphenation indicator character *c*.
.hw *word1 ...*	ignored	—	—	Exception words.
.hy *N*	on	previous	E	Hyphenate. *N* = mode.
.ie *c anything*	—	—	u	If portion of if-else; all above forms (like **if**).
.if *c anything*	—	—		If condition *c* true, accept *anything* as input, for multi-line use \{*anything* \}.
.if **!***c anything*	—	—		If condition *c* false, accept *anything*.
.if *N anything*	—	—	u	If expression $N > 0$, accept *anything*.
.if **!***N anything*	—	—	u	If expression $N \leq 0$, accept *anything*.
.if *'string1 'string2 ' anything*	—	—		If *string1* identical to *string2*, accept *anything*.
.if **!** *'string1 'string2 ' anything*	—	—		If *string1* not identical to *string2*, accept *anything*.
.ig *yy*	—	*.yy*=..	—	Ignore until call of *yy*.
.in ±*N*	N=0	previous	B,E,m	Indent.
.it *N xx*	—	off	E	Set an input-line count trap.
.lc *c*	.	none	E	Leader repetition character.
.lg *N*	on	on	—	Ligature mode on if $N>0$.
.ll ±*N*	6.5 in	previous	E,m	Line length.
.ls *N*	N=1	previous	E	Output *N*–1 *V*s after each text output line.
.lt ±*N*	6.5 in	previous	E,m	Length of title.
.mc *c N*	—	off	E,m	Set margin character *c* and separation *N*.

Summary of troff and nroff Requests

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.mk *R*	none	internal	D	Mark current vertical place in register *R*.
.na	adjust	—	E	No output line adjusting.
.ne *N*	—	*N*=1*V*	D,v	Need *N* vertical space (*V* = vertical spacing).
.nf	fill	—	B,E	No filling or adjusting of output lines.
.nh	hyphenate	—	E	No hyphenation.
.nm ±*N M S I*	off	—	E	Number mode on or off, set parameters.
.nn *N*	—	*N*=1	E	Do not number next *N* lines.
.nr *R* ±*N M*	—	—	u	Define and set number register *R*; auto-increment by *M*.
.ns	space	—	D	Turn no-space mode on.
.nx *filename*	—	end-of-file	—	Next file.
.os	—	—	—	Output saved vertical distance.
.pc *c*	%	off	—	Page number character.
.pi *program*	—	—	—	Pipe output to *program* (*nroff* only).
.pm *t*	—	all	—	Print macro names and sizes. If *t* present, print only total of sizes.
.ps ±*N*	10-point	previous	E	Point size, also \s±*N*.†
.pl ±*N*	11 in	11 in	v	Page length.
.pn ±*N*	N=1	ignored	—	Next page number is *N*.
.po ±*N*	0: 26/27 in	previous	v	Page offset.
.rd *prompt*	—	*prompt*=BEL	—	Read insertion.
.rn *xx yy*	—	ignored	—	Rename request, macro, or string *xx* to *yy*.

Summary of troff and nroff Requests

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.rm *xx*	—	ignored	—	Remove request, macro, or string.
.rr *R*	—	—	—	Remove register *R*.
.rs	—	—	D	Restore spacing. Turn no-space mode off.
.rt ±*N*	none	internal	D,v	Return *(upward only)* to marked vertical place.
.so *filename*		—	—	Interpolate contents of source file *name* when so encountered.
.sp *N*	—	*N*=1*V*	B,v	Space vertical distance *N in either direction.*
.ss *N*	12/36 em	ignored	E	Space-character size set to *N*/36 em.†
.sv *N*	—	*N*=1*V*	v	Save vertical distance *N*.
.ta *Nt* ...	0.8: 0.5in	none	E,m	Tab settings: *left* type, unless *t*=R(right), or C(centered).
.tc *c*	none	none	E	Tab repetition character.
.ti ±*N*	—	ignored	B,E,m	Temporary indent.
.tl *'left'center'right'*	—	—		Three-part title.
.tm *string*	—	newline	—	Print *string* on terminal (UNIX standard message output).
.tr *abcd....*	none	—	O	Translate *a* into *b*, *c* into *d*, etc. on output.
.uf *F*	Italic	Italic	—	Underline font set to *F* (to be switched to by ul).
.ul *N*	off	*N*=1	E	Underline *N* input lines (italicize in *troff*).
.vs *N*	1/6in:12pts	previous	E,p	Vertical base line spacing (*V*).
.wh *N xx*	—	—	v	Set location trap. Negative is with respect to page bottom.

† Point size changes have no effect in ***nroff***.

‡ The use of " ´ " as the control character (instead of " . ") suppresses the break function.

Table B-1: Notes in the Tables

Note	*Explanation*
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.
v	Default scale indicator — if not specified, scale indicators are ***ignored.***
p	Default scale indicator — if not specified, scale indicators are ***ignored.***
m	Default scale indicator — if not specified, scale indicators are ***ignored.***
u	Default scale indicator — if not specified, scale indicators are ***ignored.***

Appendix C

Escape Sequences for Characters, Indicators, and Functions

Note: The escape sequences \\, \., \", \$, *, \a, \n, \t, and \(newline) are interpreted in *copy mode* (see the section on macros, diversion, and traps).

Section Reference	*Escape Sequence*	*Meaning*
	\\	\ (to prevent or delay the interpretation of \ )
	\e	Printable version of the *current* escape character.
	\´	´ (acute accent); equivalent to \(aa
	\`	` (grave accent); equivalent to \(ga
	\−	− Minus sign in the *current* font
	\.	Period (dot) (see **de**)
	\(space)	Unpaddable space-size space character
	\0	Digit width space
	\\|	1/6 em narrow space character (zero width in *nroff*)
	\^	1/12 em half-narrow space character (zero width in *nroff*)
	\&	Non-printing, zero width character
	\!	Transparent line indicator
	\"	Beginning of comment
	\$*N*	Interpolate argument 1≤*N*≤9
	\%	Default optional hyphenation character
	\(*xx*	Character named *xx*
	**x*, *(*xx*	Interpolate string *x* or *xx*
	\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 *nroff*)
	\f*x*,\f(*xx*,\f*N*	Change to font named *x* or *xx*, or position *N*
	\h ´*N* ´	Local horizontal motion; move right *N (negative left)*
	\k*x*	Mark horizontal *input* place in register *x*
	\l ´*Nc* ´	Horizontal line drawing function (optionally with *c*)
	\L ´*Nc* ´	Vertical line drawing function (optionally with *c*)
	\n*x*,\n(*xx*	Interpolate number register *x* or *xx*

Section Reference	*Escape Sequence*	*Meaning*
	\o *'abc... '*	Overstrike characters *a, b, c, ...*
	\p	Break and spread output line
	\r	Reverse 1 em vertical motion (reverse line in ***nroff***)
	\s*N*, **\s**±*N*	Point-size change function
	\t	Non-interpreted horizontal tab
	\u	Reverse (up) 1/2 em vertical motion (1/2 line in ***nroff***)
	\v *'N '*	Local vertical motion; move down *N* ***(negative up)***
	\w *'string '*	Interpolate width of ***string***
	\x *'N '*	Extra line-space function ***(negative before, positive after)***
	\z*c*	Print *c* with zero width (without spacing)
	\{	Begin conditional input
	\}	End conditional input
	****(newline)	Concealed (ignored) newline
	*****X*	*X*, any character ***not*** listed above

Appendix D

Predefined Number Registers

Table D-1: General Number Registers

Section Reference	*Register Name*	*Description*
	c.	Input line-number in current input file; same as ..c.
	%	Current page number.
	ct	Character type (set by ***width*** 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 ***input*** line.
	ln	Output line number.
	mo	Current month (1-12).
	nl	Vertical position of last printed text base-line.
	sb	Depth of string below base line (generated by ***width*** function).
	st	Height of string above base line (generated by ***width*** function).
	yr	Last two digits of current year.

Table D-2: Read-Only Number Registers

Section Reference	*Register Name*	*Description*
	.$	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.
	.L	Current line-spacing parameter (**ls**).
	.P	1 if current page is printed, otherwise zero.
	.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 using **\x** *'N'*.
	.c	Number of ***lines*** read from current input file.

Section Reference	*Register Name*	*Description*
	.d	Current vertical place in current diversion; equal to **nl**, if no diversion.
	.f	Current font as physical quadrant (1-4).
	.h	Text base-line high-water mark on current page or diversion.
	.i	Current indent.
	.j	Current adjustment mode and type.
	.k	Horizontal text portion size of current output line.
	.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.

Appendix E

Description of *troff* Output Codes

As we mentioned before, ***troff*** is geared up to produce binary codes for a phototypesetter called a C/A/T. This appendix describes the codes for the C/A/T in detail. This information is for people who want to translate C/A/T codes for other purposes.

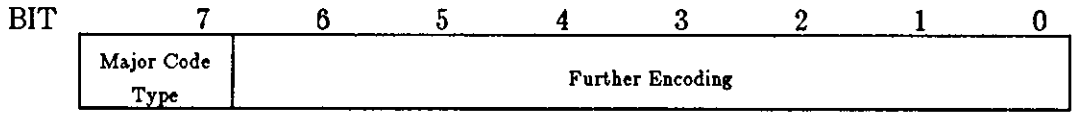
The basic mechanism of the C/A/T typesetter is a revolving drum divided into four quadrants. On each quadrant of the drum you can mount a strip of film — one strip of film corresponds to a font. Each font has 102 characters in it. Characters are exposed on the final photographic paper by 'flashing' a light through the appropriate position of the film strip on the drum. The actual font to be used is selected (as you will see later) by a combination of 'rail', 'mag', and 'font-half' — the terms 'rail' and 'mag' are hangovers from very old hot-lead typesetting technology and have no place in electro-mechanical systems, but they were carried over because typesetters can't handle new things. Point size changes are handled in the C/A/T by a series of magnifying lenses.

The C/A/T's basic unit of length (machine unit) is 1/432 inch (there are six of these units to a typesetter's 'point'). The quantum of horizontal motion is one unit. The quantum of vertical motion is three units (1/144 inch or half a point). ***troff*** uses the same system of units in its internal computations.

The C/A/T phototypesetter is driven by sending it a sequence of one-byte (eight-bit byte) codes to specify characters, fonts, point sizes, and other information. The encoding scheme used was obviously designed by someone wanting to send the minimum amount of information across a communications channel at the expense of doing vast amounts of work in the computer driving the typesetter.

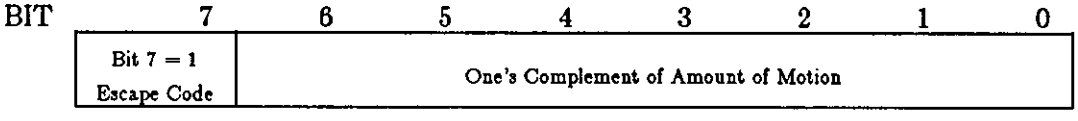
A complete C/A/T file is supposed to start with an ***initialize*** code (described later), followed by an ***escape-16*** code, then the body of the text destined for the C/A/T. The whole file ends with 14 inches of trailer, followed by a ***stop*** code. In practice, looking at ***troff***'s output file has generated disagreements on what the file really looks like, but we don't have a C/A/T around to really try it out.

Bit 7 of a code byte classifies the byte into one of two major types:

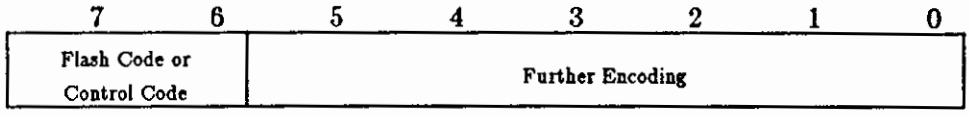


The top bit (bit 7) is encoded thus:

1 — An ***Escape Code***, specifying horizontal motion, as described below.

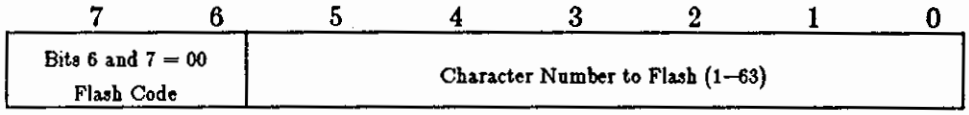


0 — indicates that bits 7 and 6 are used to further encode the code byte, as follows:

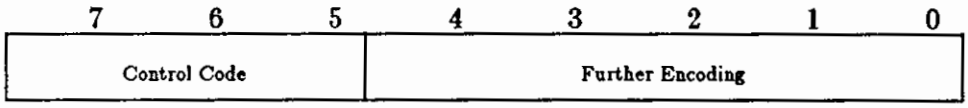


The two upper bits have these meanings:

00 — A *Flash Code*, which selects a character out of a font, as described below.



01 — A *Control Code*, which is then ***further*** encoded into one of two categories depending on whether the ***next*** bit is a one or a zero:



1 — This is a ***lead code***, described below, or

0 — in which case the control code is ***further*** encoded into one of three categories of:

- Initialization and termination.
- Selecting fonts.
- Specifying the direction of motion for escapes and leading.

We have finally reached the end of this encoding scheme. The following sections discuss each type of code in detail.

E.1. Codes 00xxxxxx — Flash Codes to Expose Characters

A code with the bits six and seven equal to zero (00xxxxxx) is a ***flash code***. A flash code specifies flashing one of 63 characters — the lower six bits of the flash code specify which character to flash. This is not enough charcater combinations to select even all the characters within a single font (there are 108 charcaters per font) and so there are control codes (described later) to select the font and which half of the font. Given that a specific font is selected via the ***rail***, ***mag***, and (for the eight-font C/A/T) the ***tilt*** codes, you then select an upper-font-half or a lower-font-half. The lower-font-half is the first 63 characters of the font, and the upper-font-half is the remaining 45 characters of the font. A flash code of greater than 46 in the upper-half of the font is considered illegal.

E.2. Codes 1xxxxxxx — Escape Codes Specifying Horizontal Motion

A code with bit seven equal to 1 (1xxxxxxx) is an ***escape code***. An ***escape code*** specifies horizontal motion. The C/A/T is a boustrophedonic device — that is, it can move in both directions, and so the direction of motion is specified by one of the control codes described later on. The

amount of horizontal motion is specified by the one's complement of the lower seven bits of the escape code.

E.3. Codes 011xxxxx — Lead Codes Specifying Vertical Motion

A codes with the top three bits equal to 011 is a ***lead code***. A ***lead code*** is a subset of the control codes in that the top three bits are 011. Such a code specifies vertical motion. The amount of the vertical motion is specified by the one's complement of the lower five bits, in vertical quanta. 'Lead' is a typesetter's term deriving from the days of hot-lead machines — the terminology sticks with us because the industry moves slowly.

E.4. Codes 0101xxxx — Size Change Codes

A byte with the top four bits equal to 0101 is a ***size-change*** code. Such a code specifies movement of a lens turret and a doubler lens to change the point size of the characters. The size-change codes are as follows:

Table E-1: Size Change Codes

Point-Size	*Binary Code*	*Octal Code*	*Point-Size*	*Binary Code*	*Octal Code*
6	0101 1000	0130	16	0101 1001	0131
7	0101 0000	0120	18	0101 0110	0126
8	0101 0001	0121	20	0101 1010	0132
9	0101 0111	0127	22	0101 1011	0133
10	0101 0010	0122	24	0101 1100	0134
11	0101 0011	0123	28	0101 1101	0135
12	0101 0100	0124	36	0101 1110	0136
14	0101 0101	0125			

Changes in size using the doubler lens change the horizontal position on the page:

If you change from:	*Follow the change with:*
Single to double	A forward escape of 55 quanta
Double to single	A reverse escape of 55 quanta

Single Point-Sizes versus Double Point-Sizes	
Single	*Double*
6	16
7	20
8	22
9	24
10	28
11	36
12	
14	
18	

E.5. Codes 0100xxxx — Control Codes

A byte with the top four bits equal to 0100 is a ***control code***. Not all of the control codes have meaning to the typesetter. The control codes are in three classes, namely:

- Initialization and termniation.
- Selecting fonts.
- Specifying the direction of motion for escapes and leading. The control codes and their meanings are:

Table E-2: C/A/T Control Codes and their Meanings

Category	*Meaning*	*Binary Code*	*Octal Code*
Initializing and Terminating	Initialize	0100 0000	0100
	Stop	0100 1001	0111
Selecting Fonts	Upper Rail	0100 0010	0102
	Lower Rail	0100 0001	0101
	Upper Mag	0100 0011	0103
	Lower Mag	0100 0100	0104
	Tilt Up	0100 1110	0116
	Tilt Down	0100 1111	0117
	Upper Font Half	0100 0110	0106
	Lower Font Half	0100 0101	0105
Specifying Direction Of Motion	Escape Forward	0100 0111	0107
	Escape Backward	0100 1000	0110
	Lead Forward	0100 1010	0112
	Lead Backward	0100 1100	0114

Note that ***tilt up*** and ***tilt down*** are ***uniplemented op-codes*** on the four-font C/A/T. However, the illustrious hackers at Berkeley implemented a program called ***rvcat*** to drive the Versatec or

the Varian printers, and they used the 0116_8 code to mean 'multiply the next lead-code by 64' to avoid having enormous runs of small lead-codes.

E.6. How Fonts are Selected

Fonts are selected by a combination of ***rail***, ***mag***, and ***tilt***. The ***tilt*** codes exist only on the eight-font C/A/T and this is the only difference between the two machines that is visible to the user. The standard version of ***troff*** doesn't know about the eight-font machine — University of Illinois is one of the places that hacked over ***troff*** to make it understand the eight-font C/A/T. The correspondence between ***rail***, ***mag***, and ***tilt*** codes is shown in this table:

Table E-3: Correspondence Between Rail, Mag, Tilt, and Font Number

Rail	*Mag*	*Tilt*	*Four-Font*	*Eight-Font*
Lower	Lower	Up	1	1
Lower	Lower	Down	1	2
Upper	Lower	Up	2	3
Upper	Lower	Down	2	4
Lower	Upper	Up	3	5
Lower	Upper	Down	3	6
Upper	Upper	Up	4	7
Upper	Upper	Down	4	8

E.7. Initial State of the C/A/T

For those wishing to write postprocessors to hack over C/A/T codes, here is the initial state of the beast:

Attribute	*Initial State*
Escape	Forward
Lead	Forward
Font-Half	Lower
Rail	Lower
Mag	Lower
Tilt	Down

READER COMMENT SHEET

Dear Customer,
We who work here at Sun Microsystems wish to provide the best possible documentation for our products. To this end, we solicit your comments on this manual. We would appreciate your telling us about errors in the content of the manual, and about any material which you feel should be there but isn't.

Typographical Errors:
Please list typographical errors by page number and actual text of the error.

Technical Errors:
Please list errors of fact by page number and actual text of the error.

Content:
Please list errors of fact by page number and actual text of the error.

Content:

Did this guide meet your needs? If not, please indicate what you think should be added or deleted in order to do so. Please comment on any material which you feel should be present but is not. Is there material which is in other manuals, but would be more convenient if it were in this manual?

Layout and Style:

Did you find the organization of this guide useful? If not, how would you rearrange things? Do you find the style of this manual pleasing or irritating? What would you like to see different?