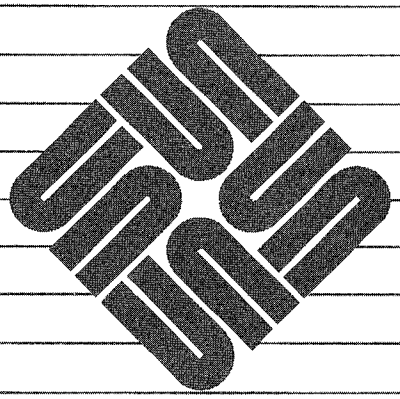
sun®
microsystems

Using NROFF & TROFF



Part Number: 800-1755-11
Revision A of 27 March, 1990

Material in this manual comes from a number of sources: *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; *Writing Papers with Nroff Using -me*, Eric P. Allman, University of California, Berkeley; and *Introducing the UNIX System*, Henry McGilton, Rachel Morgan, McGraw-Hill Book Company, 1983. These materials are gratefully acknowledged.

Contents

Chapter 1 Introduction **1**
1.1. nroff and troff 1
Text Formatting Versus Word Processing 2
The Evolution of nroff and troff 3
Preprocessors and Postprocessors 4
1.2. troff, Typesetters, and Special-Purpose Formatters 4
1.3. Using the nroff and troff Text Formatters 4
Options Common to nroff and troff 5
Options Applicable Only to nroff 5
Options Applicable Only to troff 6
1.4. General Explanation of troff and nroff Source Files 6
Backspacing 7
Comments 7
Continuation Lines 8
Transparent Throughput 8
Formatter and Device Resolution 8
Specifying Numerical Parameters 8
Numerical Expressions 10
1.5. Output and Error Messages 11

Chapter 2 Line Format **13**
2.1. Controlling Line Breaks 14
.br — Break Lines 16
Continuation Lines and Interrupted Text 16

2.2. Justifying Text and Filling Lines 17
.ad — Specify Adjusting Styles 17
.na — No Adjusting 18
.nf and .fi — Turn Filling Off and On 19
2.3. Hyphenation 20
.nh and .hy — Control Hyphenation 20
.hw — Specify Hyphenation Word List 21
.hc — Specify Hyphenation Character 22
2.4. .ce — Center Lines of Text 23
2.5. .ul and .cu — Underline or Emphasize Text 24
2.6. .uf — Underline Font 25

Chapter 3 Page Layout **27**
3.1. Margins and Indentations 29
.po — Set Page Offset 29
.ll — Set Line Length 29
.in — Set Indent 30
.ti — Temporarily Indent One Line 32
3.2. Page Lengths, Page Breaks, and Conditional Page Breaks 35
.pl — Set Page Length 35
.bp — Start a New Page 35
.pn — Set Page Number 36
.ne — Specify Space Needed 36
3.3. Multi-Column Page Layout by Marking and Returning 37
.mk — Mark Current Vertical Position 37
.rt — Return to Marked Vertical Position 38

Chapter 4 Line Spacing and Character Sizes **39**
4.1. .sp — Space Vertically 39
4.2. .ps — Change the Size of the Type 40
4.3. .vs — Change Vertical Distance Between Lines 42
4.4. .ls — Change Line Spacing 43
4.5. \x Function — Get Extra Line-Space 44

4.6. .sv — Save Block of Vertical Space 44
4.7. .os — Output Saved Vertical Space 45
4.8. .ns — Set No Space Mode 45
4.9. .rs — Restore Space Mode 45
4.10. .ss — Set Size of Space Character 46
4.11. .cs — Set Constant-Width Characters 46

Chapter 5 Fonts and Special Characters **47**
5.1. .ft — Set Font 48
5.2. .fp — Set Font Position 49
5.3. .fz — Force Font Size 49
5.4. .bd — Artificial Boldface 50
5.5. Character Set 51
5.6. Fonts 52
5.7. .lg — Control Ligatures 52

Chapter 6 Tabs, Leaders, and Fields **55**
6.1. .ta — Set Tabs 55
Setting Relative Tab Stops 56
Right-Adjusted Tab Stops 56
Centered Tab Stops 56
.tc — Change Tab Replacement Character 57
Summary of Tabs 58
6.2. Leaders — Repeated Runs of Characters 59
.lc — Change the Leader Character 61
6.3. .fc — Set Field Characters 62

Chapter 7 Titles and Page Numbering **67**
7.1. Titles in Page Headers 67
7.2. Fonts and Point Sizes in Titles 69
7.3. .pc — Page Number Character 70
7.4. .tl Request — Three Parameters 71

Chapter 8 troff Input and Output **73**

8.1. .so — Read Text from a File 73
8.2. .nx — Read Next Source File 75
8.3. Pipe Output to a Specified Program (nroff only) 75
8.4. .rd — Read from the Standard Input 76
8.5. .ex — Exit from nroff or troff 78
8.6. .tm — Send Messages to the Standard Error File 78

Chapter 9 Strings **79**
9.1. .ds — Define Strings 80
9.2. .as — Append to a String 81
9.3. Removing or Renaming String Definitions 83

Chapter 10 Macros, Diversions, and Traps **85**
10.1. Macros 85
.de — Define a Macro 85
.rm — Remove Requests, Macros, or Strings 87
.rn — Rename Requests, Macros or Strings 88
Macros With Arguments 88
.am — Append to a Macro 92
Copy Mode Input Interpretation 92
10.2. Using Diversions to Store Text for Later Processing 92
.di — Divert Text 93
.da — Append to a Diversion 94
10.3. Using Traps to Process Text at Specific Places on a Page 94
.wh — Set Page or Position Traps 95
.ch — Change Position of a Page Trap 96
.dt — Set a Diversion Trap 96
.it — Set an Input-Line Count Trap 96
.em — Set the End of Processing Trap 97

Chapter 11 Number Registers **99**
11.1. .nr — Set Number Registers 99
11.2. Auto-Increment Number Registers 101

11.3. Arithmetic Expressions with Number Registers 102
11.4. .af — Specify Format of Number Registers 103
11.5. .rr — Remove Number Registers 105

Chapter 12 Drawing Lines and Characters **107**
12.1. \u and \d Functions — Half-Line Vertical Movements 107
12.2. Arbitrary Local Horizontal and Vertical Motions 108
\v Function — Arbitrary Vertical Motion 108
\h Function — Arbitrary Horizontal Motion 109
12.3. \0 Function — Digit-Size Spaces 110
12.4. '\ ' Function — Unpaddable Space 112
12.5. \| and \^ Functions — Thick and Thin Spaces 112
12.6. \& Function — Non-Printing Zero-Width Character 113
12.7. \o Function — Overstriking Characters 114
12.8. \z Function — Zero Motion Characters 115
12.9. \w Function — Get Width of a String 116
12.10. \k Function — Mark Current Horizontal Place 117
12.11. \b Function — Build Large Brackets 117
12.12. \r Function — Reverse Vertical Motions 119
12.13. Drawing Horizontal and Vertical Lines 119
\l Function — Draw Horizontal Lines 119
\L Function — Draw Vertical Lines 120
Combining the Horizontal and Vertical Line Drawing Functions 121
12.14. .mc — Place Characters in the Margin 121

Chapter 13 Character Translations **123**
13.1. Input Character Translations 123
13.2. .ec and .eo — Set Escape Character or Stop Escapes 123
13.3. .cc and .c2 — Set Control Characters 124
13.4. .tr — Output Translation 124

Chapter 14 Automatic Line Numbering **125**
14.1. .nm — Number Output Lines 125

14.2. `.nn` — Stop Numbering Lines 126

Chapter 15 Conditional Requests **127**
15.1. `.if` — Conditional Request 127
15.2. `.ie` and `.el` — If-Else and Else Conditionals 130
15.3. `.ig` — Ignore Input Text 130

Chapter 16 Debugging Requests **133**
16.1. `.pm` — Display Names and Sizes of Defined Macros 133
16.2. `.fl` — Flush Output Buffer 134
16.3. `.ab` — Abort 134

Chapter 17 Environments **135**
17.1. `.ev` — Switch Environment 135

Appendix A `troff` Request Summary **137**

Appendix B Font and Character Examples **143**
B.1. Font Style Examples 143
B.2. Non-ASCII Characters and *minus* on the Standard Fonts 144
B.3. Non-ASCII Characters and ´, `, G, +, –, =, and * on the Special Font 144

Appendix C Escape Sequences **147**

Appendix D Predefined Number Registers **149**

Appendix E `troff` Output Codes **151**
E.1. Codes `00`*xxxxxx* — Flash Codes to Expose Characters 152
E.2. Codes `1`*xxxxxxx* — Escape Codes Specifying Horizontal Motion 153
E.3. Codes `011`*xxxxx* — Lead Codes Specifying Vertical Motion 153
E.4. Codes `0101`*xxxx* — Size Change Codes 153
E.5. Codes `0100`*xxxx* — Control Codes 154
E.6. How Fonts are Selected 155

E.7. Initial State of the C/A/T 155

Index **157**

Tables

Table 1-1 Scale Indicators for Numerical Input 9

Table 1-2 Default Scale Indicators for Certain `troff` Requests and Functions 9

Table 1-3 Arithmetic Operators and Logical Operators for Expressions 10

Table 2-1 Constructs that Break the Filling Process 15

Table 2-2 Formatter Requests that Cause a Line Break 16

Table 2-3 Adjusting Styles for Filled Text 17

Table 5-1 Exceptions to the Standard ASCII Character Mapping 52

Table 6-1 Types of Tab Stops 58

Table 7-1 Requests that Cause a Line Break 69

Table 11-1 Access Sequences for Auto-incrementing Number Registers 102

Table 11-2 Arithmetic Operators and Logical Operators for Expressions 102

Table 11-3 Interpolation Formats for Number Registers 104

Table 12-1 `troff` Width Function — `ct` Number Register Values 117

Table 12-2 Pieces for Constructing Large Brackets 118

Table 15-1 Built-In Condition Names for Conditional Processing 129

Table A-1 Summary of `nroff` and `troff` Requests 137
Table A-2 Notes in the Tables 142

Table B-1 Summary of `troff` Special Characters 144

Table C-1 `troff` Escape Sequences 147

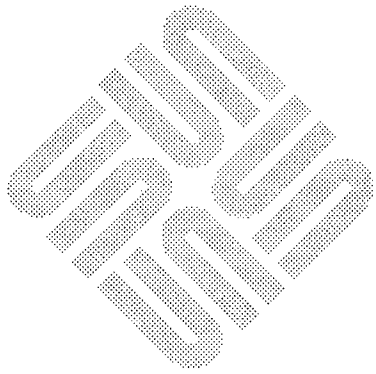
Table D-1 General Number Registers 149
Table D-2 Read-Only Number Registers 149

Table E-1 Size Change Codes 153
Table E-2 Single Point-Sizes versus Double Point-Sizes 154
Table E-3 C/A/T Control Codes and their Meanings 154
Table E-4 Correspondence Between Rail, Mag, Tilt, and Font Number 155

Figures

Figure 2-1 Filling and Adjusting Styles 18

Figure 3-1 Layout of a Page 28



Preface

This manual provides reference information and examples for the text formatters `nroff` and `troff`. We assume you are familiar with a terminal keyboard and the Sun system. If you are not, see *SunOS User's Guide: Getting Started* for information on the basics, like logging in and the Sun file system. If you are not familiar with text editors, read *SunOS User's Guide: Doing More* and the chapter "Introduction to Text Editing" in *Editing Text Files*. Finally, we assume that you are using a Sun Workstation, although specific terminal information is also provided.

For additional details on Sun system commands and programs, see the *SunOS Reference Manual.*

Summary of Contents

Here is a summary of the chapters that follow:

1. *Introduction* — Describes what `troff` can do for you, some tools you can use with `troff` or `nroff` to refine your results, how to use `nroff` and `troff`, the differences between the two text formatting programs, and a little about the mechanisms built-in to `nroff` and `troff`.
2. *Line Format* — Explains how the text formatting programs fill and adjust text input lines and how various formatting requests affect filling and adjusting functions in `troff`.
3. *Page Layout* — Describes the default page layout parameters built-in to `troff` and how you can alter them. Also explains how certain formatting requests interact in laying out pages.
4. *Line Spacing and Character Sizes* — Explains the available type and spacing sizes in `troff` and `nroff`, and how to change them.
5. *Fonts and Special Characters* — Describes the fonts available with `nroff` and `troff` and how to change them.
6. *Tabs, Leaders, and Fields* — Explains what tabs, leaders, and fields are, and how to set them.
7. *Titles and Page Numbering* — Explains how to create page headers and page footers. Also covers how to use the built-in `troff` page number register to print page numbers on your document automatically.

8. `troff` Input and Output — Describes how to embed files within files, to switch input from one file to another, to display a message on your terminal when `troff` reaches a certain point in a file, and in `nroff` only, how to pipe the output from a file to a program by using a special `nroff` command in the file.
9. *Strings* — Explains how to give a string of characters a new name so you can reference them easily. Also provides a facility for referencing the values of the strings.
10. *Macros, Diversions, and Traps* — Describes how to define macros, store information in diversions, and use diversions and traps to process text at specific places on pages.
11. *Number Registers* — Explains what `troff` number registers are and what you can use their values for.
12. *Drawing Lines and Characters* — Describes the several built-in `troff` functions for moving to arbitrary places on the page and for drawing things.
13. *Character Translations* — Describes how to change the escape character and translate the value of one character into another.
14. *Automatic Line Numbering* — Explains how to use the `troff` requests for numbering lines in the output file.
15. *Conditional Requests* — Describes `troff` mechanisms for conditionally accepting input.
16. *Debugging Requests* — Explains requests for displaying names and sizes of defined macros, flushing the output buffer, and aborting the formatting.
17. *Environments* — Describes how to shift input processing between the three `nroff/troff` environments.

A. `troff` *Request Summary* — A quick reference summarizing `nroff` and `troff` requests.

B. *Font and Character Examples* — Several tables of special characters like Greek letters, foreign punctuation, and math symbols.

C. *Escape Sequences* — Summarizes escape sequences for obtaining values of number registers, for describing arbitrary motions and drawing things, and for specifying certain miscellaneous functions.

D. *Predefined Number Registers* — Tables of `troff` General and Predefined Number Registers

E. `troff` *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. **`Boldface 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.

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- or two-character arguments are indicated by a pair of lower case letters. Character string arguments are indicated by multi-character mnemonics.

1

Introduction

1.1. 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;
- 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 manual 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.[1]

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

[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.

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 indent paragraphs and number pages 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 manual at all, but rather the chapter "Formatting Documents with the -ms Macros" in the *Formatting Documents* manual. If, on the other hand, you would like to learn the fine details of a programming 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 — 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 when using troff is not to use it directly, but through some intermediary such as one of the macro packages, or one of the various preprocessors described in *Formatting Documents*. In the few cases where existing macro 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 existing packages. In accordance with this strategy 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. If you are interested in the complete story, look into the troff source itself.

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 boldface by typing control indicators. The word processor usually displays these activated features using inverse video or special marks on the screen. 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 WYSIWYG and pronounced 'wizzi-wig'). Unfortunately, as has been pointed out, the problem with many WYSIWYG 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 usually 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 perform any editing — its only job is reading text from a file and formatting that text for printing on some device. Entering the text into the file, and formatting the text from that file for printing are two separate and independent operations. You prepare your file of text using a text 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 many documents were written using the capabilities of `nroff` or `troff`.

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 phototypesetting 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.

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*. Refer to the manual *Formatting Documents* for explanations of nroff and troff pre- and postprocessors.

1.2. troff, Typesetters, and Special-Purpose Formatters

Please be sure to read this : this section covers some aspects of troff that are generally glossed over in the traditional UNIX system 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.

1.3. 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 discusses the formatting commands. Then you run the formatter at the 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 -ms 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 –msun macro package.

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;* 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*.

-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*
Adds the macro file `/usr/lib/tmac/tmac.`*name* before 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.

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.

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 elapsed time.

1.4. General Explanation of `troff` and `nroff` Source Files

This section of the `nroff` and `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=πr^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 five 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. 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 aesthetic 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.

To print the escape character (usually backslash), use `\e` (backslash e).

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.

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.
\" 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.

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
```

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.

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. Refer to Chapter 10 for information describing diversions.

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).

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 1-1 *Scale Indicators for Numerical Input*

Scale Indicator	*Meaning*	*Number of basic units* `troff`	`nroff`
`i`	Inch	432	240
`c`	Centimeter	432×50/127	240×50/127
`P`	Pica = 1/6 inch	72	240/6
`m`	Em = *S* points	6×*S*	*C*
`n`	En = Em/2	3×*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, *V*s for the vertically-oriented requests and functions, *p* for the vertical spacing request; and *u* for the number register and conditional requests. See Table 1-2 for a summary of the default scale indicators for the `troff` requests and functions that take scale indicators.

Table 1-2 *Default Scale Indicators for Certain* `troff` *Requests and Functions*

Request	*Default Scaling Unit*	*Request*	*Default Scaling Unit*
`.ll`	ems	`.pl`	vertical units (Vs)
`.in`	"	`.wh`	"
`.ti`	"	`.ch`	"
`.ta`	"	`.dt`	"
`.lt`	"	`.sp`	"
`.po`	"	`.sv`	"
`.mc`	"	`.ne`	"
`\h`	"	`.rt`	"
`\l`	"	`\v`	"
`.nr`	machine units (u)	`\x`	"
`.if`	"	`\L`	"
`.ie`	"	`.vs`	picas (p)

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 form, but the parameter finally stored is rounded to an integer number of basic units.

The *absolute position* indicator | (the pipe character) may precede a number *N* to generate the absolute distance to the vertical or horizontal place *N*. For vertically-oriented requests and functions, | *N* becomes the absolute distance in basic units from the current vertical place on the page or in a *diversion* (see Chapter 10 for 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.

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 1-3 *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.

1.5. Output and Error Messages

The output from `.tm`, `.pm`, and the prompt from `.rd`, as well as various *error* messages are written onto the *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.

2

Line Format

Perhaps the most important reason for using `troff` or `nroff` is to use its filling and adjusting capabilities. Here is 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` (space size) 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 baseline position on the page for this line is in the `nl` number register. The text baseline high-water mark on the current page is in the `.h` number register.

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 2-1 *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 (break) 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 2-2 *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 (break) request at the appropriate place, as described below.

.br — Break Lines

The .br (break) 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*	
Mnemonic:	break
Form of Request:	.br
Initial Value:	Not Applicable
If No Argument:	cause break
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.

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.

2.2. Justifying Text and Filling Lines

.ad — Specify Adjusting Styles

To change the style of text justification, use the .ad (adjust) request to specify one of the four different methods for adjusting text:

Table 2-3 *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	
Mnemonic:	adjust
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 Table 2-3.
Notes:	E (see Table A-2)

The current adjustment indicator *c* can be obtained from the .j number register.

The following figure 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` (adjust both) request, or a `.ad n` (adjust normal) 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` (adjust left) request, which has the same effect as using a `.na` (no adjust) 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` (adjust right) 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` (adjust center) request.

Figure 2-1 *Filling and Adjusting Styles*

`.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` (no adjust) request. Output lines are still filled, providing that filling hasn't also been turned off — see the `.nf` (no fill) and `.fi` (fill) requests below. If filling is still on, `troff` or `nroff` produces flush left, ragged right output. To turn adjusting back on (return to the previous state), use the `.ad` request.

Summary of the `.na` *Request*	
Mnemonic:	no adjust
Form of Request:	`.na`
Initial Value:	Adjusting is on by default
If No Argument:	adjusting 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. To turn adjusting back on (return to the previous state), use the `.ad` request.
Notes:	E (see Table A-2)

`.nf` and `.fi` — Turn Filling Off and On

The `.nf` (no fill) 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` (fill) request turns on filling. 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*	
Mnemonic:	fill
Form of Request:	`.fi`
Initial Value:	Filling is on by default
If No Argument:	filling is turned on
Explanation:	Fill subsequent output lines. The number register `.u` is 1 in fill mode and 0 in nofill mode.
Notes:	E,B (see Table A-2)

Summary of the .nf *Request*	
Mnemonic:	no fill
Form of Request:	.nf
Initial Value:	Filling is on by default
If No Argument:	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 number register .u is 1 in fill mode and 0 in nofill mode.
Notes:	E,B (see Table A-2)

2.3. 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 that 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. We should emphasize that, although the examples show text that is both filled and justified, it is during filling that troff or nroff hyphenates words, not adjusting.

If you have words in your input text containing hyphens (such as jack-in-the-box, or co-worker), troff or nroff will, if necessary, split these words over two lines, even if hyphenation is turned off.

.nh and .hy — Control Hyphenation

Normally, when you invoke troff or nroff, hyphenation is turned on, but you can change this. The .nh (no hyphenation) request turns off automatic hyphenation. When hyphenation is turned off, the only words that are split over more than one line are those that already contain hyphens. Hyphenation can be turned on again with the .hy (hyphenate) request.

You can give .hy 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 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***	
Mnemonic:	no hyphenation
Form of Request:	`.nh`
Initial Value:	Hyphenation is on by default
If No Argument:	hyphenation is turned off
Explanation:	Turn automatic hyphenation off.
Notes:	E (see Table A-2)

Summary of the `.hy` ***Request***	
Mnemonic:	hyphenation
Form of Request:	`.hy` *N*
Initial Value:	Hyphenation is on by default in mode 1.
If No Argument:	$N=1$.
Explanation:	Turn automatic hyphenation on for $N \geq 1$, or off for $N=0$. If $n=1$, all words are subject to hyphenation. 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. Note: odd values of N (except 1) don't make sense.
Notes:	E (see Table A-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` (hyphenate words) 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 in 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*	
Mnemonic:	hyphenate word
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.

.hc — Specify Hyphenation Character

A *hyphenation indicator* character may be embedded in a word to specify desired hyphenation points, or may precede the word to suppress hyphenation. For example, hyphenation looks particularly disruptive if it occurs in titles. So, if you had a long title like:

Input and Output Conventions and Character Translations,

you could shorten it, or you could insert the hyphenation character just before the first character of each of the long words at the end of the title. The input might look like this:

```
.H C "Input and Output Conventions and \%Character \%Translations"
```

(If you are using a reasonable line length, you don't need to worry about hyphenation occurring earlier in the title in this example.)

Here is an example of using the hyphenation character to specify acceptable hyphenation points within a word. The word "workstation" is often mis-hyphenated because of the collection of consonants at the end of "work" and the beginning of "station". So, your input might look like this:

```
work\%station
```

Summary of the .hc *Request*	
Mnemonic:	hyphenation character
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 (see Table A-2)

2.4. .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 (center) request, which centers 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*	
Mnemonic:	center
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 (see Table A-2)

2.5. .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 (underline) 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
following line of text
```

simply underlines the following line of text. Unlike .ce, though, .ul does not turn filling off. 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.

Summary of the `.ul` *Request*	
Mnemonic:	underline
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 (see Table A-2)

Another form of underlining is called up with the `.cu` request, and asks for continuous underlining. 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.

Summary of the `.cu` *Request*	
Mnemonic:	continuously underline
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 (see Table A-2)

2.6. `.uf` — Underline Font

`nroff` automatically underlines characters in the *underline* font, specifiable with a `.uf` (underline font) request. The underline font is normally Times Italic and is mounted on font position 2. In addition to the `.ft` (font) request and the `\f`*F*, the underline font may be selected by the `.ul` (underline) request and the `.cu` (continuous underline) request. Underlining is restricted to an output-device-dependent subset of *reasonable* characters.

Summary of the `.uf` *Request*	
Mnemonic:	underline font
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).

3

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 following figure 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.

Figure 3-1 *Layout of a Page*

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 could have the left margin moved, not by indenting, but by changing the page-offset via a `.po +0.5i` request. Now *all* text would be moved to the left, and because the line-length hasn't changed, the right margin would move as well. The example can't show this because page offset is measured from the margin, and because this example is in a box, changing the page offset within the box is meaningless.

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 'bulleted' 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 rightward, 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, note that tab stops are relative to the current left margin as we show here with a couple of blocks of text with different indents. Note that the positions of the tab stops are shown with exclamation point (`!`) 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

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.

.po — Set Page Offset

The usable 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.

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' 8.5×11-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` (indent) requests and `.ti` (temporary indent) 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` (line length) request is discussed below.

Summary of the .po Request	
Mnemonic:	page offset
Form of Request:	`.po` ±*N*
Initial Value:	0 in `nroff`, 26/27 inch in `troff`.
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 usually 8.5 inches. In `nroff` the initial page-offset is zero.
Notes:	v (see Table A-2)

The current page-offset is available in the `.o` register.

.ll — Set Line Length

`troff` gives you full control over the length of the printed lines. By the way, typographers don't use terms like 'line-length', they use the word 'measure' to mean the length of a line. They always measure vertical distances in 'picas'.

Nevertheless, to set the line-length in `troff`, use the `.ll` (line length) 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 C/A/T typesetter was 7.54 inches, by the way. To use the full width, you 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 (page offset) 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 Chapter 7, *Titles and Page Numbering*) is independent of the line-length set by the .ll request — the length of a three-part title is set by the .lt request.

Summary of the .ll Request	
Mnemonic:	line length
Form of Request:	.ll ±*N*
Initial Value:	6.5 inches
If No Argument:	Previous value
Explanation:	Set the line-length to *N* where *N* 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 usually 8.5 inches.
Notes:	E, m (see Table A-2)

.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 (indent) request discussed here, and via the .ti (temporary indent) request described below.

The .in (indent) 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 only the first line of a paragraph indented, you don't use the .in request, but you use the

`.ti` (temporary indent) 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 leftward:

```
.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 +2.0i` makes lines two inches longer, whereas `.ll 2.0i` 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. So, in the above example, the lines:

[2] *Petronius Arbiter*, A.D. 60.

```
.ll +0.5i
.in -0.5i
```

could have been

```
.ll
.in
```

and would have had the same effect.

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.

Summary of the .in Request	
Mnemonic:	indent
Form of Request:	`.in` ±*N*
Initial Value:	0
If No Argument:	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. The `.in` request causes a *break*.
Notes:	E, m (see Table A-2)

`.ti` — Temporarily Indent One Line

The `.ti` (temporary indent) 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 specified units just like this one. Three of what? The default unit for the `.ti` request, as for most horizontally-oriented requests — `.ll` (line length), `.in` (indent), and `.po` (page offset) — 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
\(butab the \fL\&.po\fP 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
\(butab the \fL\&.in\fP request sets the
indent from the left margin for all following text.
.ti -0.2i                         move left margin back temporarily
\(butab the \fL\&.ti\fP 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.

Remember 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	
Mnemonic:	temporary indent
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. The .ti request causes a *break*.
Notes:	E, m (see Table A-2)

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 is done automatically.

If you want top and bottom margins 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). Refer to Chapter 10 for more information on diversions.

`.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 won't need to use the `.pl` request because the standard setting is right for all but the most esoteric purposes.

Summary of the `.pl` *Request*	
Mnemonic:	page length
Form of Request:	`.pl` ±*N*
Initial Value:	11 inches
If No Argument:	11 inches
Explanation:	Set page length to ±*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` number register.
Notes:	v (see Table A-2)

`.bp` — Start a New Page

This request causes a break and skips to a new page.

Summary of the .bp *Request*	
Mnemonic:	begin page
Form of Request:	.bp ±*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 ±*N* is given, the new page number will be ±*N*. Also see the .ns (no space) request. The .bp request causes a *break*.
Notes:	v (see Table A-2)

.pn — Set Page Number

Summary of the .pn *Request*	
Mnemonic:	page number
Form of Request:	.pn ±*N*
Initial Value:	*N*=1
If No Argument:	Ignored
Explanation:	The next page (when it occurs) will have the page number ±*N*. A .pn request 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.

.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 achieve this ranging from simple to complicated. One of the simplest ways is to use the .ne (need) vertical space 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*	
Mnemonic:	need
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 (see Table A-2)

3.3. Multi-Column 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 useful 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.

`.mk` — Mark Current Vertical Position

Summary of the `.mk` *Request*	
Mnemonic:	mark
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 the `.rt` request.

.rt — Return to Marked Vertical Position

Summary of the .rt *Request*	
Mnemonic:	return
Form of Request:	.rt ±*N*
Initial Value:	Not applicable
If No Argument:	return to place marked by a previous .mk request.
Explanation:	Return *upward only* to a marked vertical place in the current diversion. If ±*N* (with respect to the current place) is given, the place is ±*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 chapter *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 .mk *R* . . . .sp ~ \n*R*u.

4

Line Spacing and Character Sizes

4.1. .sp — Space Vertically

You get extra vertical space with the .sp (space) 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).

Summary of the .sp *Request*	
Mnemonic:	space
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 (see Table A-2)

4.2. .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 (point size) 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.

6 point: Pack my box with five dozen liquor jugs.
7 point: Pack my box with five dozen liquor jugs.
8 point: Pack my box with five dozen liquor jugs.
9 point: Pack my box with five dozen liquor jugs.
10 point: Pack my box with five dozen liquor jugs.
11 point: Pack my box with five dozen liquor jugs.
12 point: Pack my box with five dozen liquor jugs.
14 point: Pack my box with five dozen liquor jugs.
16 point: Pack my box with five dozen liquor jugs.
18 point: Pack my box with five dozen liquor jugs.
20 point: Pack my box with five dozen liquor jugs.
22 point: Pack my box with five dozen liquor jugs.
24 point: Pack my box with five dozen liquor jugs.
28 point: Pack my box with five dozen liquor
36 point: Pack my box with five doz

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 size) 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.

The current size is available in the `.s` number register. `nroff` ignores type size control.

Summary of the .ps *Request*	
Mnemonic:	point size
Form of Request:	.ps ±*N*
Initial Value:	10 points
If No Argument:	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. The sequence .ps +*N* .ps *N* works the same as .ps +*N* .ps –*N* because the previous requested value is also remembered. Ignored in nroff.
Notes:	E (see Table A-2)

4.3. .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 (vertical spacing) 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 13 points between baselines. Typographers call this '11 on 13', so when you hear some one say that a book is set in '11 on 13', you know that it's 11-point type with 13-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, both `.ps` and `.vs` revert to the previous size and vertical spacing respectively.

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` number register.

Summary of the `.vs` ***Request***	
Mnemonic:	vertical spacing
Form of Request:	`.vs` *N*
Initial Value:	1/6 inch in `nroff`, 12 points in `troff`.
If No Argument:	Previous value
Explanation:	Set vertical base-line spacing size *V*. Transient *extra* vertical space available with `\x'`*N*`'` (see section on `\x` Function).
Notes:	E, p (see Table A-2)

4.4. `.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*	
Mnemonic:	line spacing
Form of Request:	.ls *N*
Initial Value:	*N*=1
If No Argument:	Previous value
Explanation:	Set *line* spacing to ±*N*. *N*–1 *V*s *(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 (see Table A-2)

4.5. \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 embedded 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.

4.6. .sv — Save Block of Vertical Space

A block of vertical space is ordinarily requested using the .sp (space) 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 .sv *Request*	
Mnemonic:	save space
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 (see Table A-2)

4.7. .os — Output Saved Vertical Space

Summary of the .os *Request*	
Mnemonic:	output saved space
Form of Request:	.os
Initial Value:	Not applicable
If No Argument:	Output saved vertical space
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.

4.8. .ns — Set No Space Mode

Summary of the .ns *Request*	
Mnemonic:	no-space mode
Form of Request:	.ns
Initial Value:	Not applicable
If No Argument:	Turn on no-space mode
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 (see Table A-2)

4.9. .rs — Restore Space Mode

Summary of the .rs *Request*	
Mnemonic:	restore space mode
Form of Request:	.rs
Initial Value:	Not applicable
If No Argument:	Turn off no-space mode
Explanation:	Restore spacing — turn off no-space mode.
Notes:	D (see Table A-2)

4.10. .ss — Set Size of Space Character

Summary of the .ss *Request*	
Mnemonic:	space-character size
Form of Request:	.ss *N*
Initial Value:	12/36 em
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 (see Table A-2)

4.11. .cs — Set Constant-Width Characters

Summary of the .cs *Request*	
Mnemonic:	constant spacing
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 (see Table A-2)

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 B, *Font and Character Examples*.

`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`.

5.1. .ft — Set Font

Summary of the .ft *Request*	
Mnemonic:	font
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 (see Table A-2)

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. Refer to Chapter 2 for a more detailed description of the .ul request.

Fonts can also be changed within a line or word with the in-line request \f:

bold*face* text

is produced by the input

```
\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 font position 1, italic (I) on position 2, bold (B) on position 3, and special (S) on position 4 — the mnemonic 'R I B S' might help you remember.

5.2. .fp — Set Font Position

Summary of the .fp *Request*	
Mnemonic:	font position
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 that 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.

5.3. .fz — Force Font Size

Summary of the .fz *Request*	
Mnemonic:	font size
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` causes 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.

There is also a way to get 'synthetic' bold fonts by overstriking letters with a slight offset. Look at the `.bd` request.

5.4. .bd — Artificial Boldface

Summary of the .bd *Request*	
Mnemonic:	bold
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 (see Table A-2)

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 \(*x, where *x* represents an upper- or lower-case Roman letter reminiscent of the Greek. Thus to get

Σ(α×β) → ∞

in raw 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 toward*
\(if	∞	*infinity*

A complete list of these special names occurs in Appendix B, *Font and Character Examples*.

In `eqn`, explained in the chapter "Formatting Mathematics with `eqn`" in *Formatting Documents*, you can achieve the same effect 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. For example, 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`.

5.5. 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 B, *Font and Character Examples*. 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 B. The three ASCII exceptions are mapped as follows:

Table 5-1 *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 B. The ASCII characters @, #, ", ´, `, <, >, \, {, }, ~, ^, and _ exist *only* on the Special Font and are printed as a one-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.

5.6. 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 and others 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*x*, \f(*xx*, 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- or two-character font name or the numerical font position, 1 through 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.

5.7. .lg — Control Ligatures

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 Chapter 12 the section on "Arbitrary Horizontal Motion" (the `\h` function) for some examples on constructing these ligatures.

Summary of the `.lg` ***Request***	
Mnemonic:	ligature
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`.

6

Tabs, Leaders, and Fields

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.

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` (tab) 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 exclamation 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!
```

When we format the above example, we get this output:

```
!              !              !              !              !
word-one       word-two       word-three     word-four      word-five
!              !              !              !              !
```

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 !
```

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
July tab 5
August tab 9
September tab 15
October tab 60
November tab 85
December tab 126
.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.

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
Month tab Shipments
.ta 2.0iR
July tab 5
August tab 9
September tab 15
October tab 60
November tab 85
December tab 126
.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`" in *Formatting Documents*. 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.

`.tc` — Change Tab Replacement Character

A tab inserts blank spaces 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` (tab character) 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 sections "Drawing Vertical Lines" and "Drawing Horizontal Lines" in the chapter "Arbitrary Motions and Drawing Lines and Characters."

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 chapter "Arbitrary Motions and Drawing Lines and Characters."

Summary of the .tc *Request*	
Mnemonic:	tab character
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 (see Table A-2)

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 6-1 *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*	
Mnemonic:	tab
Form of Request:	`.ta` *Nt* ...
Initial Value:	0.8 inches 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 (see Table A-2)

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 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:

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:

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 solid line:

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:

```
.DS
.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"
.DE
```

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

.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:

```
.DS
.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"
.DE
```

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 added before the leaders as space. 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	
Mnemonic:	leader character
Form of Request:	.lc *c*
Initial Value:	.
If No Argument:	Removed — successive \as act like tabs
Explanation:	The leader repetition character becomes *c*, or is removed. Successive leader requests (\as) act like tabs.
Notes:	E (see Table A-2)

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 parceling 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 (field characters) 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 formatting:

```
!                                   !
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 to #
                            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 to #
                            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 parceled out so that the sub-fields are uniformly left-adjusted, right-adjusted, or centered between the current position and the next tab stop:

```
.ta 5.0i                        set a single tab at five inches
.fc # @                         set field delimiter character to #
                                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 5.0i
.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	
Mnemonic:	field character
Form of Request:	`.fc` *f p*
Initial Value:	Field mechanism is off
If No Argument:	Field mechanism is turned off.
Explanation:	Set the field delimiter to *f*; set the padding indicator to *p* (if specified) or to the *space* character if *p* is not specified. In the absence of arguments, the field mechanism is turned off.

7

Titles and Page Numbering

7.1. Titles in Page Headers

This is an area where things get tougher, because `troff` doesn't do any of this 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 the title).
- 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 the title parameters), 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
```

See Chapter 10 for a more detailed description of the .wh request. No dot (.) is used before NP in the when request because in this case, we're specifying the name of a macro, not calling a macro. 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, 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 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 7-1 *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` (break) request at the appropriate place.

7.2. Fonts and Point Sizes in Titles

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.

Summary of the `.lt` ***Request***	
Mnemonic:	length of title
Form of Request:	`.lt` ±*N*
Initial Value:	6.5 inches
If No Argument:	Previous value
Explanation:	Set length of title to ±*N*. The line-length and the title-length are *independent*. Indents do not apply to titles; page-offsets do.
Notes:	E, m (see Table A-2)

There are several ways to fix the problems of point sizes and fonts in titles. For

the simplest applications, we can define the .NP macro 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 Chapter 17, *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.

7.3. .pc — Page Number Character

You can change the page number character with the .pc request.

Summary of the .pc Request	
Mnemonic:	page-number character
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 %.

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`.

7.4. `.tl` Request — Three Parameters

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:

Hunting the Snark – 71 – Smiles and Soap

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***	
Mnemonic:	title
Form of Request:	`.tl` *'left'center'right'*
Initial Value:	Nothing
If No Argument:	Nothing
Explanation:	The strings in the *left*, *center*, and *right* fields 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.

8

troff Input and Output

We now describe two troff requests that we omitted earlier, because their usefulness is more apparent when you understand the troff command line. Normally troff 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 troff requests embedded in the document text.

8.1. .so — Read Text from a File

The .so request, which tells troff to switch over and take its source 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 troff command line:

```
hostname% troff macros document
hostname%
```

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:

```
hostname% troff document
hostname%
```

The first thing troff 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*, troff 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 troff 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 `troff` command line:

```
hostname% troff document | lpr
hostname%
```

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):

```
hostname% 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
hostname%
```

It is obviously much easier to format the whole thing with a `troff` command line like this:

```
hostname% troff book | lpr
hostname%
```

than it would be if you had to supply all the filenames in the right order. Notice that we used the comment feature of `troff` to tie chapter titles to filenames.

Summary of the .so *Request*	
Mnemonic:	source
Form of Request:	.so *filename*
Explanation:	Switch source file — the top input (file reading) level is switched to *filename*. The sourced-in file is read directly and processed immediately when the .so line is encountered. When the new file ends, input is again taken from the original file. .sos may be nested.

8.2. .nx — Read Next Source File

Summary of the .nx *Request*	
Mnemonic:	next
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*. There is no return to the file containing the .nx command.

8.3. Pipe Output to a Specified Program (nroff only)

A couple of examples of programs you might want you pipe your nroff output to are lpr and col. Your source line might look like this:

```
.pi /usr/ucb/lpr
```

or

```
.pi /usr/bin/col
```

if you had formatted tables in your source file.

Summary of the .pi *Request*	
Mnemonic:	pipe
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*.

8.4. .rd — Read from the Standard Input

Another troff request that switches input from the file you specify is the .rd (read) request. The *standard input* can be the user's keyboard, a pipe, or a file. The .rd request reads an insertion from the standard input. When troff encounters the .rd request, it prompts for input by sounding the terminal bell or flashing the screen. 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:

```
hostname% troff valentine | lpr
who:Peter

hostname%
```

After typing the name Peter you have to press the RETURN key twice, since troff needs a blank line to end input. The result 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 troff command again:

```
hostname% troff valentine | lpr
who:Bill

hostname%
```

and again for Joe, and for Manuel, and Louis, and Alphonse, and ...

Since troff 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 .rd Request	
Mnemonic:	read
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*. Use the standard way to access arguments in macros (see Chapter 10.

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 (see the previous section); the process would ultimately be ended by a .ex in the insertion file. Example:

Letter File	*Names File*
Dear ...	John
.rd	*blank line*
.	Bill
.	*blank line*
.	.ex
.nx Letter	

To put everything together, you could use:

```
hostname% cat Names | troff Letter
```

8.5. .ex — Exit from nroff or troff

Summary of the .ex *Request*	
Mnemonic:	exit
Form of Request:	.ex *prompt*
Explanation:	Exit from nroff or troff. Text processing is terminated exactly as if all input had ended.

8.6. .tm — Send Messages to the Standard Error File

The .tm (terminal message) 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 longer 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*	
Mnemonic:	terminal message
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.

9

Strings

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. (See Chapter 12 for more detailed information on drawing lines and characters.

Fortunately, `troff` provides a way that 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 in the text by the string definition.

A string is a named sequence 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` (define string) request. You can later add more characters to the end of the string by using the `.as` (append to string) request.

String names may be either one or two characters long. 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 named *x*, and the more complicated notation:

```
\*(xx
```

for a two-character string named *xx*.

String references and macro invocations may be nested.

9.1. .ds — Define Strings

You create a string (and define its initial value) with the .ds (define string) request. The line

```
.ds e \o"e\'"
```

defines the string e to have the value \o"e\'"

You refer to them with the sequence * *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 in Chapter 5, *Fonts and Special Characters*, we noted that troff doesn't know about the Scandinavian ligatures — you have to decide for yourself how to define them. Here are our 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 entitled "\h Function — Arbitrary Horizontal Motion" in Chapter 12 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 character 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, the backslash and the newline characters are disregarded resulting in the next line being added to the current one. So you can make a long string simply by ending each line except 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*	
Mnemonic:	define string
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.

9.2. `.as` — Append to a String

The `.as` (append to string) 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 (section number) 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 says:

> .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

9.

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:

9.2.0.0.0.

which, if you look, should be the section number of the stuff you are now reading.

Summary of the `.as` ***Request***	
Mnemonic:	append to string
Form of Request:	`.as` *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.

9.3. Removing or Renaming String Definitions

Strings (just like macros) can be renamed with the `.rn` (rename) request, or can be removed from the namelist with the `.rm` (remove) request. Refer to Chapter 10 for more detailed descriptions of the `.rn` and `.rm` commands.

10

Macros, Diversions, and Traps

10.1. Macros

Before we can go much further in `nroff` or `troff`, we need to learn something about the macro facility. In its simplest form, a macro is just shorthand notation similar to a string. A macro is a collection of several separate `troff` commands which, when bundled together, achieves (sometimes complex) formatting when the macro is invoked. Whereas a string is somewhat limited because its definition is specific, a macro can interpret 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. Macros are created by `.de` and `.di` requests, and appended to by `.am` and `.da` requests; `.di` and `.da` requests 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*. 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. String references and macro invocations may be nested. Any of these entities may be renamed with a `.rn` request or removed with a `.rm` request.

`.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 (define) request:

```
.de PP
.sp
.ti +2m
..
```

The first line names the macro (we used .PP) which is a standard macro 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 the beginning and end of the definition, 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 should be greater, the vertical space should be less, and the font should be Roman. Instead of changing the whole document, we need only change the definition of the .PP macro to something like

```
.de PP  \" paragraph macro
.sp 2p
.ti +3m
.ft R
..
```

and the change takes effect everywhere we used .PP.

The notation \" 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).

Summary of the .de *Request*	
Mnemonic:	define
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 '. .'.

.rm — Remove Requests, Macros, or Strings

Summary of the .rm *Request*	
Mnemonic:	remove
Form of Request:	.rm *xx*
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.

.rn — Rename Requests, Macros or Strings

Summary of the .rn *Request*	
Mnemonic:	rename
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 removed first.

Refer to Chapter 9, *Strings* for information on defining strings.

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 an incremental amount: .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 half an inch, then it is only necessary to change the definitions of .BS and .BE, not the whole paper.

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 can 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 the arguments 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≤*N*≤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 backslash (\). 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.

Let's 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 UNIX
```

will produce UNIX.

The definition of .SM is

```
.de SM
\s-2\\$1\s+2
..
```

Within a macro definition, the symbol \\$*n* refers to the *nth* 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  UNIX  ),
```

produces

UNIX),

while

```
.SM  UNIX  ).  (
```

produces

(UNIX).

It is convenient to reverse the order of arguments because trailing punctuation is much more common than leading.

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 discuss in the section *Copy Mode Input Interpretation*. 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 like the ones in this manual, 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 the 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 called. If that's what you want, fine, but that isn't the case 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:

```
.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 for left title, center title, and right title, respectively. 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.

.am — Append to a Macro

Summary of the .am *Request*	
Mnemonic:	append to macro
Form of Request:	.am *xx yy*
Initial Value:	Not applicable
If No Argument:	.yy=. .
Explanation:	Append to macro *xx* (append version of .de).

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 preceded by backslash (\ newline) are eliminated.
- Comments indicated by \" are eliminated.
- \t and \a are interpreted as ASCII horizontal tab and SOH respectively (see Chapter 6, *Tabs, Leaders, and Fields* for more information).
- \\ is interpreted as \
- \. is interpreted as "."

These interpretations can be suppressed by adding another \ (backslash) to the beginning of the command. 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.

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.

.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 It.XX. 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 chapter *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 baseline (`.h` register), and the current diversion name (`.z` register).

Summary of the .di Request	
Mnemonic:	divert
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 (see Table A-2)

`.da` — Append to a Diversion

Summary of the .da Request	
Mnemonic:	append to diversion
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.

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` (when) request at any page position including the top. This trap position may be changed using the `.ch` (change) 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.

`.wh` — Set Page or Position Traps

Summary of the `.wh` *Request*	
Mnemonic:	when
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* is 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 (see Table A-2)

.ch — Change Position of a Page Trap

Summary of the .ch *Request*	
Mnemonic:	change trap
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 (see Table A-2)

.dt — Set a Diversion Trap

Summary of the .dt *Request*	
Mnemonic:	diversion trap
Form of Request:	.dt *N xx*
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 *xx*. Another .dt will redefine the diversion trap. If no arguments are given, the diversion trap is removed.
Notes:	D, v (see Table A-2)

.it — Set an Input-Line Count Trap

Summary of the .it *Request*	
Mnemonic:	input-line-count trap
Form of Request:	.it *N xx*
Initial Value:	Not applicable
If No Argument:	Turn off trap
Explanation:	Set an input-line-count trap to invoke the macro *xx* 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 in-line or trap-invoked macros.
Notes:	E (see Table A-2)

.em — Set the End of Processing Trap

Summary of the .em *Request*	
Mnemonic:	end macro
Form of Request:	.em *xx*
Initial Value:	Not applicable
If No Argument:	No trap installed
Explanation:	Call the macro *xx* when all input has ended. The effect is the same as if the contents of *xx* had been at the end of the last file processed.

11

Number Registers

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` (number register) 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 D). In addition, users may define their 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 listed in 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 (see Table D-1) for a complete list); 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 a `.nr` request because they are "read only" (see Table D-2) for a complete list).

11.1. `.nr` — Set Number Registers

You create and modify number registers using the `.nr` (number register) request. In its simplest form, the `.nr` request 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 Macros" in *Formatting Documents*). 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 10
.nr VS 12
```

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, *, \$, 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*	
Mnemonic:	number register
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 (see Table A-2)

11.2. 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) section number register that starts off with the value 0 and is incremented by 1 every time you use it. This might be applicable (for instance) to numbering the sections of a document automatically — something you might expect a computer to do for you. You might also define a numbered list macro that would clock up the item number every time you added a new list 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 11-1 *Access Sequences for Auto-incrementing Number Registers*

Request	*Access Sequence*	*Effect on Register*	*Value Interpolated*
`.nr x N M`	`\n`*x*	none	*N*
`.nr xx N M`	`\n(`*xx*	none	*N*
`.nr x N M`	`\n+`*x*	*x* incremented by *M*	*N+M*
`.nr x N M`	`\n-`*x*	*x* decremented by *M*	*N–M*
`.nr xx N M`	`\n+(`*xx*	*xx* incremented by *M*	*N+M*
`.nr xx N M`	`\n-(`*xx*	*xx* decremented by *M*	*N–M*

11.3. 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 the value in the PS macro 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 11-2 *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.

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.

11.4. `.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` (assign format) 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 11-3 *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, ... aa, ab, ... aaa, ...
A	upper-case letters	0, A, B, C, ... AA, AB, ... AAA, ...

Summary of the `.af` *Request*	
Mnemonic:	assign format
Form of Request:	`.af` *R c*
Initial Value:	Arabic
If No Argument:	Ignored
Explanation:	Assign format *c* to register *R*.

11.5. `.rr` — Remove Number Registers

If you create many number registers dynamically, you may 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` (remove register) request:

```
.rr xy
```

removes the *xy* number register from the list.

Summary of the `.rr` *Request*	
Mnemonic:	remove register
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.

12

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.

12.1. \u and \d Functions — Half-Line Vertical Movements

If you can't or don't want to use `eqn`, subscripts and superscripts are then most easily done with the half-line local motions `\u` (for up) and `\d` (for down). To move up the page half a point, insert a `\u` at the desired place, and to go down the page half a point, 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 restores the point-size to the previous value after the superscript. This example input:

```
... area of a circle is 'Area = \(*pr\u\s-22\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* vertical spacing, 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.

12.2. Arbitrary Local Horizontal and Vertical Motions

The next two sections describe the in-line \v (vertical) and the \h (horizontal) local motion 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 *to the right* and *down*.

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, be zero.

\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.

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.

`\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 backward 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

[3] Omar Khayyám — *the Rubáiyát*

```
\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 in the chapter *Fonts and Special Characters*, 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 backward 0.4 of the width of the letter 'a'.
`e`	Letter 'e' in the string.

12.3. `\0` Function — Digit-Size 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    \0Description
.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.

12.4. '\ ' Function — Unpaddable Space

There is also the in-line \ function, which is the \ character (backslash) 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 appears on one line, you would type it in as

```
\ \ \f(LBnroff\ -Tlp\fP\ \fImyfile\fP\ \
```

in-line in the text.

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 to the right, they 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.

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 here.

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

système 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

∉.

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, troff centers the characters on the widest of them, 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 in the section "\b Function — Build Large Brackets" later in this chapter.

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 (string top) and sb (string bottom) 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 (character type) is set to a value between 0 and 3:

Table 12-1 `troff` *Width Function* — `ct` *Number Register Values*

`ct` Number Register Value	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.

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 to its absolute (hence, the |) beginning (\kx*word*\h'\nxu) plus 2 machine units (+2u) and overprinting it, resulting in

word

12.11. `\b` Function — Build Large Brackets

The Special (mathematical) font contains a number of 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 12-2 *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 bracketing) 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 (1/2-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:

$$\left\{\left[\ x\ \right]\right\}$$

by typing this:

```
\b'\(lt\(lk\(lb' \b'\(lc\(lf' x \b'\(rc\(rf' \b'\(rt\(rk\(rb'
```

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`.

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.

`\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.

`\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 of the (optional) *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.

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 baseline 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.

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 \s12\(br\s0
```

request (that is, place a 12-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.

Currently, this request is not bug-free, and the margin character only appears to the right of the right margin, but not in left margins. Also, you'll notice that the marginal bars do not appear on incomplete lines, such as this one.

Summary of the .mc *Request*	
Mnemonic:	margin character
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.
Notes:	E, m (see Table A-2)

13

Character Translations

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` (translate) request (refer to the section entitled `.tr` — *Output Translation*). All others are ignored.

13.2. `.ec` and `.eo` — Set Escape Character or Stop Escapes

The escape character \ introduces *escape sequences* — meaning the following character is something else, 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 changed with an `.ec` (escape character) 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` (escape off) request and restored with the `.ec` request.

Summary of the `.ec` ***Request***	
Mnemonic:	escape character
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***	
Mnemonic:	escape mechanism off
Form of Request:	`.eo`
Initial Value:	Escape mechanism is on
If No Argument:	Turn escape mechanism off.
Explanation:	Turn escape mechanism off.

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*	
Mnemonic:	control character
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*	
Mnemonic:	no-break control character
Form of Request:	.c2 *c*
Initial Value:	′
If No Argument:	′
Explanation:	Set the *no-break* control character to *c*, or reset to ' ′ '.

13.4. .tr — Output Translation

One character can be made a stand-in for another character using the .tr (translate) 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*	
Mnemonic:	translate
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 (see Table A-2)

14

Automatic Line Numbering

14.1. .nm — Number Output Lines

Output lines may be numbered automatically via the .nm (number) request. Refer to the following table for a summary of the .nm request. When in effect, a three-digit, Arabic 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 digit spaces. Blank lines, other vertical spaces, and lines generated by .tl are *not* numbered. Numbering can be temporarily suspended with the .nn (no number) request (see below), or with an .nm followed by a later .nm +0. In addition, a line number indent *I*, and the number-text separation *S* may be specified in digit-spaces. Further, it can be specified that only those line numbers that are multiples of some number *M* are to be printed (the others will appear as blank number fields).

Summary of the .nm Request	
Mnemonic:	numbering
Form of Request:	.nm ± *N M S I*
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. *N* is the line number counter (or incrementer if you use ±*N*), *M* is the multiple of the numbered lines to be printed on the page, *S* is the spacing between line numbers and text, and *I* is the amount of indent for the line numbers. 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 (see Table A-2)

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 (no number) request.

Summary of the .nn Request	
Mnemonic:	no numbering
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 (see Table A-2)

As an example, the paragraph portions of this chapter are numbered with *M*= 3: .nm 1 3 was placed at the beginning of the chapter; .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

Conditional Requests

15.1. .if — Conditional Request

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 perform 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 a .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*	
Mnemonic:/if, if-else, else	
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* is 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 15-1 *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 \}.

15.2. .ie and .el — If-Else and Else Conditionals

The request .ie (if-else) is almost 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. Refer to the *Summary of the* .if *Requests* for summaries of .ie and .el.

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.

15.3. .ig — Ignore Input Text

Another mechanism for conditionally accepting input text is via the .ig (ignore) 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*	
Mnemonic:	ignore
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 `..` if no argument is specified on the request. `.ig` behaves exactly 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.

16

Debugging Requests

`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.

16.1. .pm — Display Names and Sizes of Defined Macros

The .pm (print macros) 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 (that it's swallowing your entire file).

Summary of the .pm Request	
Mnemonic:	print macros
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 are given in *blocks* of 128 characters.

16.2. .fl — Flush Output Buffer

The .fl (flush) request flushes the output buffer — this can be used when you're using nroff interactively.

Summary of the .fl Request	
Mnemonic:	flush
Form of Request:	.fl
Initial Value:	Not applicable
If No Argument:	adjusting is turned off
Explanation:	Flush output buffer. Used in interactive debugging to force output.

16.3. .ab — Abort

A final useful request in the debugging category is the .ab (abort) request which basically bails out and stops the formatting.

Summary of the .ab Request	
Mnemonic:	abort
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.

17

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.

17.1. `.ev` — Switch Environment

The command `.ev` *n* shifts to environment *n*; *n* must be in the range 0 through 2. A `.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	
Mnemonic:	environment
Form of Request:	`.ev` *N*
Initial Value:	*N*=0
If No Argument:	Switch back to previous environment
Explanation:	Switch to environment *N*, where 0≤*N*≤2. Switching is done in push-down fashion so that restoring a previous environment must be done with `.ev` rather than specific reference.

A

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.

Table A-1 *Summary of* nroff *and* troff *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*.

Table A-1 *Summary of* nroff *and* troff *Requests— Continued*

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ce *N*	off	*N*=1	B,E	Center following *N* input text lines.
.ch *xx N*	—	—	v	Change trap location.
.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≤*N*≤4.

Table A-1 *Summary of* nroff *and* troff *Requests— Continued*

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ft *F*	Roman	previous	E	Change to font *F* = *x*, *xx*, or 1 through 4. Also \fx, \f(xx, \f*N*.
.fz *S F N*	none	—	—	Forces font *F* or *S* for special characters to be in size *N*.
.hc *c*	\%	\%	E	Hyphenation indicator character *c*.
.hw *word1 ...*	ignored	—	—	Exception words.
.hy *N*	on	previous	E	Hyphenate. *N* = mode.
.ie *c anything*	—	—	—	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*	—	—	—	If expression *N* > 0, accept *anything*.
.if **!***N anything*	—	—	—	If expression *N* ≤ 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.

Table A-1 *Summary of* nroff *and* troff *Requests— Continued*

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.lt ±*N*	6.5 in	previous	E,m	Length of title.
.mc *c N*	—	off	E,m	Set margin character *c* and separation *N*.
.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* by ±*N*; 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.

Table A-1 *Summary of* nroff *and* troff *Requests— Continued*

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.rd *prompt*	—	*prompt*=BEL	—	Read insertion.
.rn *xx yy*	—	ignored	—	Rename request, macro, or string *xx* to *yy*.
.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* equals R (right), or C (centered).
.tc *c*	space	removed	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 (to standard error).
.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).

Table A-1 *Summary of* nroff *and* troff *Requests— Continued*

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.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 A-2 *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.*

B

Font and Character Examples

B.1. 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 ¼-em 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

" ´ \ ^ _ ` ~ / < > { } # @ + − = *
α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ σ ς τ υ φ χ ψ ω
Γ Δ Θ Λ Ξ Π Σ Υ Φ Ψ Ω
√ ¯ ≥ ≤ ≡ ~ ≈ ≠ → ← ↑ ↓ × ÷ ± ∪ ∩ ⊂ ⊃ ⊆ ⊇ ∞ ∂
§ ∇ ¬ ∫ ∝ ∅ ∈ ‡ ⇒ ⇐ | ○ ⌈ ⌊ ⌉ ⌋ { } | ⌊⌋ ⌈⌉ |

B.2. Non-ASCII Characters and *minus* on the Standard Fonts

Char	Input Name	Character Name	Char	Input Name	Character Name
’	’	close quote	fi	\(fi	fi
‘	`	open quote	fl	\(fl	fl
—	\(em	3/4 Em dash	ff	\(ff	ff
-	-	hyphen or	ffi	\(Fi	ffi
-	\(hy	hyphen	ffl	\(Fl	ffl
−	\-	current font minus	°	\(de	degree
•	\(bu	bullet	†	\(dg	dagger
□	\(sq	square	′	\(fm	foot mark
_	\(ru	rule	¢	\(ct	cent sign
¼	\(14	1/4	®	\(rg	registered
½	\(12	1/2	©	\(co	copyright
¾	\(34	3/4			

B.3. 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.

Table B-1 *Summary of* troff *Special Characters*

Char	Input Name	Character Name	Char	Input Name	Character Name
+	\(pl	math plus	σ	\(*s	sigma
−	\(mi	math minus	ς	\(ts	terminal sigma
=	\(eq	math equals	τ	\(*t	tau
*	\(**	math star	υ	\(*u	upsilon
§	\(sc	section	φ	\(*f	phi
´	\(aa	acute accent	χ	\(*x	chi
`	\(ga	grave accent	ψ	\(*q	psi
_	\(ul	underrule	ω	\(*w	omega
/	\(sl	slash (matching backslash)	A	\(*A	Alpha†
α	\(*a	alpha	B	\(*B	Beta†
β	\(*b	beta	Γ	\(*G	Gamma
γ	\(*g	gamma	Δ	\(*D	Delta
δ	\(*d	delta	E	\(*E	Epsilon†
ε	\(*e	epsilon	Z	\(*Z	Zeta†
ζ	\(*z	zeta	H	\(*Y	Eta†
η	\(*y	eta	Θ	\(*H	Theta
θ	\(*h	theta	I	\(*I	Iota†
ι	\(*i	iota	K	\(*K	Kappa†

Table B-1 *Summary of* `troff` *Special Characters— Continued*

Char	*Input Name*	*Character Name*	*Char*	*Input Name*	*Character Name*
κ	\(*k	kappa	Λ	\(*L	Lambda
λ	\(*l	lambda	M	\(*M	Mu†
μ	\(*m	mu	N	\(*N	Nu†
ν	\(*n	nu	Ξ	\(*C	Xi
ξ	\(*c	xi	O	\(*O	Omicron†
o	\(*o	omicron	Π	\(*P	Pi
π	\(*p	pi	P	\(*R	Rho†
ρ	\(*r	rho	Σ	\(*S	Sigma
T	\(*T	Tau†	∞	\(if	infinity
Y	\(*U	Upsilon	∂	\(pd	partial derivative
Φ	\(*F	Phi	∇	\(gr	gradient
X	\(*X	Chi†	¬	\(no	not
Ψ	\(*Q	Psi	∫	\(is	integral sign
Ω	\(*W	Omega	∝	\(pt	proportional to
√	\(sr	square root	∅	\(es	empty set
_	\(rn	root en extender	∈	\(mo	member of
≥	\(>=	>=	|	\(br	box vertical rule
≤	\(<=	<=	‡	\(dd	double dagger
≡	\(==	identically equal	⇒	\(rh	right hand
≈	\(~=	approx =	⇐	\(lh	left hand
~	\(ap	approximates	|	\(or	or
≠	\(!=	not equal	○	\(ci	circle
→	\(->	right arrow	⎧	\(lt	left top of big curly bracket
←	\(<-	left arrow	⎩	\(lb	left bottom
↑	\(ua	up arrow	⎫	\(rt	right top
↓	\(da	down arrow	⎭	\(rb	right bot
×	\(mu	multiply	⎨	\(lk	left center of big curly bracket
÷	\(di	divide	⎬	\(rk	right center of big curly bracket
±	\(+-	plus-minus	⎪	\(bv	bold vertical
∪	\(cu	cup (union)	⎣	\(lf	left floor (left bottom of big square bracket)
∩	\(ca	cap (intersection)	⎦	\(rf	right floor (right bottom)
⊂	\(sb	subset of	⎡	\(lc	left ceiling (left top)
⊃	\(sp	superset of	⎤	\(rc	right ceiling (right top)
⊆	\(ib	improper subset	\	\e	backslash (escape charact
⊇	\(ip	improper superset			

C

Escape Sequences

Note: The escape sequences \\, \., \", \$, *, \a, \n, \t, and \(newline) are interpreted in copy mode (see Chapter 10).

Table C-1 `troff` *Escape Sequences*

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*

Table C-1 troff *Escape Sequences— Continued*

Escape Sequence	*Meaning*
\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 (default character is baseline rule in troff or underline in nroff; optionally with character *c*)
\L'*Nc*'	Vertical line drawing function (default character is box rule; optionally with character *c*)
\n*x*, \n(*xx*	Interpolate number register *x* or *xx*
\o'*abc...*'	Overstrike characters *a*, *b*, *c*, ...
\p	Break and spread output line
\r	Reverse one-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

D

Predefined Number Registers

Table D-1 *General Number Registers*

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 baseline.
`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*

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'`.

Table D-2 *Read-Only Number Registers— Continued*

Register Name	*Description*
.c	Number of *lines* read from current input file.
.d	Current vertical place in current diversion; equal to nl, if no diversion.
.f	Current font as physical quadrant (1-4).
.h	Text baseline high-water mark on current page or diversion.
.i	Current indent.
.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 (a string, not a number).

E

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.

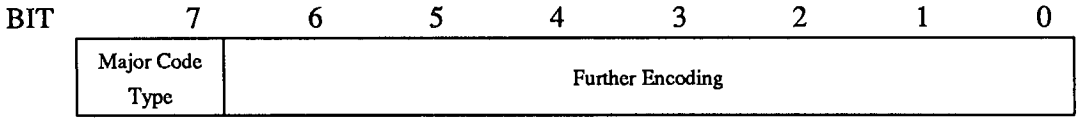
The basic mechanism of the C/A/T typesetter is a revolving drum divided into four quadrants. On each quadrant of the drum you can mount a strip of film — one strip of film corresponds to a font. Each font has 108 characters in it. Characters are exposed on the final photographic paper by 'flashing' a 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.

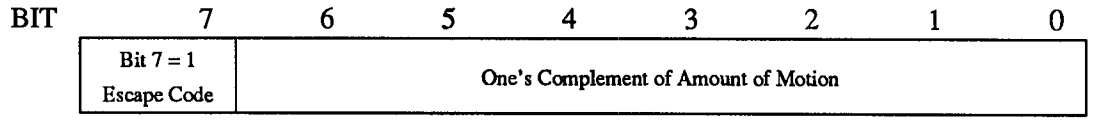
A complete C/A/T file is supposed to start with an *initialize* code (described later), followed by an *escape-16* code, then the body of the text destined for the C/A/T. The whole file ends with 14 inches of trailer, followed by a *stop* code. In practice, looking at troff's output file has generated 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:

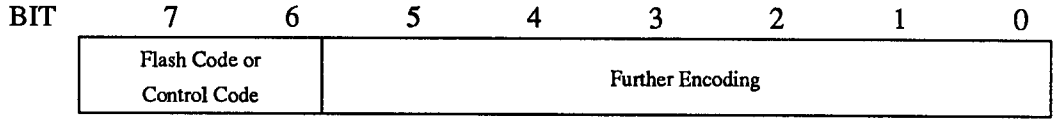


The top bit (bit 7) is encoded thus:

1 — An *Escape Code*, specifying horizontal motion, as described below.

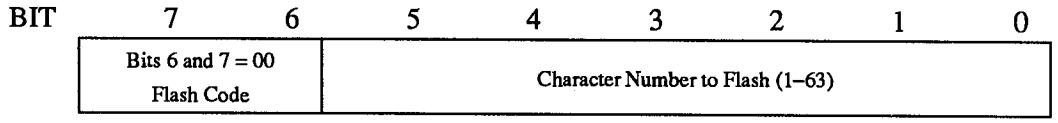


0 — indicates that bits 7 and 6 are used to further encode the code byte, as follows:

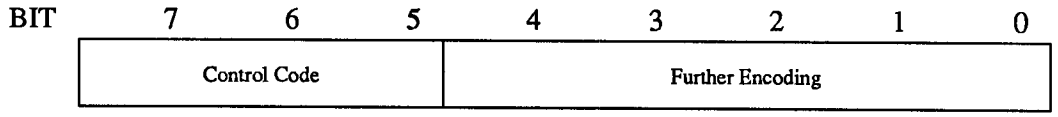


The two upper bits have these meanings:

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 00*xxxxxx* — Flash Codes to Expose Characters

A code with the bits six and seven equal to zero (00*xxxxxx*) 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 character combinations to select even all the characters within a single font (there are 108 characters 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 1*xxxxxxx* — 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 011*xxxxx* — 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 0101*xxxx* — 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

Table E-2 *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 0100*xxxx* — 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 termination.
- Selecting fonts.
- Specifying the direction of motion for escapes and leading. The control codes and their meanings are:

Table E-3 *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 *unimplemented 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-4 *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

Index

Special Characters

.$ (number of arguments) number register, **89**
\& (zero-width non-printing) function, **113**
% (page-number) number register, **36**, 99
\ (unpaddable space) function, **112**
\^ (thin space) function, **112**
\| (thick space) function, **112**

0

\0 (digit-size space) function, **110**

A

\a (leader character) function, 60
.a (post-line extra space) number register, **44**
.ab (abort) request, **134**
access format for number registers, 103
accessing strings, 80
.ad (adjust) request, **17**
adjusting, **13**
  center, 17
  flush left, ragged right, 17
  flush right, ragged left, 17
  justified, 17
.af (format of number register) request, **103**
.am (append to a macro) request, **92**
append to a
  diversion, 94
  macro, 92
  string, 81
arguments to macros, 89
arithmetic expressions with number registers, 102
.as (append to string) request, **81**
auto-incrementing number registers, 101
automatic hyphenation, 20

B

\b (bracket) function, **117**
backslash – how to print it in troff, 7
basic request, 6
.bd (boldface) request, **50**
begin page, 35
blank lines, 15
bold-face request, 50
box lines, 121
.bp (start new page) request, **35**
.br (break lines) request, **16**, 15
bracket drawing function, 117
break request, 15, 16

C

\c (continuation line) function, **16**
C/A/T codes
  control, 152
  escape, 152
  file organization, 151
  flash, **152**, 152
.c2 (set no-break control character) request, **124**
.cc (set control character) request, **124**
.ce (center lines) request, **24**, 23 *thru* 24
centered tabs, 56
.ch (change position of a trap) request, **96**
change bars, 121
change position of a trap, 96
character translation (substitution), 124
comments in troff source files, 7
concealed newlines, 8
conditional page break, 36
conditional processing of input, 127
conditional request
  .el, 129
  .ie, 129
  .if, 127
  .ig, 130
constant character space width mode request, 46
continuation lines, 8, 16
continuously underline request, 25
control character setting, 124
control code, 152
control lines in troff, 6
copy mode, 92
creating number registers, 99
.cs (set constant character space width mode) request, **46**
ct (character type) number register, **117**
.cu (continuously underline) request, **25**

D

\d (move down) function, **107**
.d (vertical place in current diversion) number register, **94**
.da (append to a diversion) request, **94**
.de (define macro) request, **85**

defining troff objects
    macros, 85
    number registers, 99
    strings, 80
deleting number registers, **105**
device resolution, 8
.di (divert text) request, **94**
diversion traps, 94, 96
diversions, 93, 94
divert text, 94
dl (width of last finished diversion) number register, **93**
dn (height of last finished diversion) number register, 93
document preparation
    formatters, 1 *thru* 11
    nroff program, 1 *thru* 11
    text formatters, 1 *thru* 11
    troff program, 1 *thru* 11
drawing in troff
    boxes, 121
    brackets, 117
    horizontal lines, 119
    vertical lines, 119, 120
.ds (define string) request, **80**
.dt (set a diversion trap) request, **96**
dy (day of month) number register, **99**

E

.ec (set escape character) request, **123**
.el (else conditional) request, **129**
.em (set the end-of-processing trap) request, **97**
end-of-file, 15
end-of-processing traps, 97
end-of-sentence, 14
environment switching, 135
.eo (set escape off) request, **123**
escape character, 123
escape code for C/A/T, 152
.ev (switch environment) request, **135**
.ex (terminal message) request, **78**
expressions with number registers, 102

F

.f (current font) number register, **52**
.fc (set field characters) request, **62**
.fi (fill) request, **19**
field character, 62
fields, 62
fill request, 19
filler character, 14
filling, **13**
.fl (flush buffer) request, **134**
flash code, **152**, 152
flush output buffer, 134
font position request, 49
footers, 67, 71
force font size request, 49
.fp (change font position) request, **49**
.ft (set font) request, **48**
.fz (force font size) request, **49**

G

general number registers
    % — page-number, **36**, 99
    ct — character type, **117**
    dl — width of last finished diversion, **93**
    dn — height of last finished diversion, 93
    dy — day of month, **99**
    mo — month of year, **99**
    nl — vertical position of last baseline, **99**, 93
    sb — string depth below baseline, **116**
    st — string height above baseline, **116**
    yr — last two digits of year, **99**
get vertical space request, 39

H

\h (horizontal motion) function, **109**
.h (text high-water mark) number register, **14**, 94
half em-space, 112
half-line motions
    \d (move down) function, 107
    \u (move up) function, 107
hanging indent, 33
hard blank, 13
.hc (hyphenation character) request, **22**
headers, 67, 71
horizontal lines, 119
horizontal motion, 109, 110, 112, 114
horizontal place marker, 117
.hw (hyphenate word) request, **21**
.hy (hyphenate) request, **20**, 21
hyphenation, **20**
    automatic, 20
    control, 20
    indicator, 21
    indicator character, 22
    special cases, 21
    specifying location, 21
    turn on and off, 20

I

.i (current indent) number register, **32**, 34
.ie (if-else conditional) request, **129**
.if (conditional processing) request, **127**
.ig (ignore lines) request, **130**
ignoring input lines, 130
.in (indent) request, **31**
in-line functions
    \ (unpaddable space) function, **112**
    \& (zero-width non-printing) function, **113**
    \^ (thin space) function, **112**
    \| (thick space) function, **112**
    \0 (digit-size space) function, **110**
    \a (leader character) function, 60
    \b (bracket) function, **117**
    \c (continuation line) function, **16**
    \d (move down) function, **107**
    \h (horizontal motion) function, **109**
    \k (mark horizontal position) function, **117**
    \l (horizontal line) function, **119**
    \L (vertical line) function, **120**, 119
    \o (overstrike) function, **114**

in-line functions, *continued*
\p (break and spread) function, **15**
\r (reverse line) function, **119**
\u (move up) function, **107**
\v (vertical motion) function, **108**
\w (width) function, **116**
\x (get extra line space) function, **44**
\z (zero motion) function, **115**
include
from file, 73
from standard input, 76
incrementing number registers, 101
indentation
first line of paragraph, 32
permanent, 31
temporary, 32
input-line-count traps, 94, 96
interpolating number registers, 99, 103
interrupted line, 16
.it (set an input-line-count trap) request, **96**
italic correction, 112
itemized lists, 33

J

.j (current adjustment indicator) number register, **17**

K

\k (mark horizontal position) function, **117**

L

\l (horizontal line) function, **119**
\L (vertical line) function, **120**, 119
.l (line-length) number register, **30**
large boxes, 121
.lc (set leader character) request, **61**
leaders and leader characters, 59, 60
left margin, 29
length of title, 69
.lg (set ligature mode) request, **53**
ligatures, **53**
line adjustment indicators
both, 17
center, 17
indentation, 31
left, 17
normal, 17
right, 17
line drawing
functions, 119, 120
horizontal, 119
vertical, 119, 120
line numbering
start, 125
suspend, 126
line spacing request, 43
line-length, 29
.ll (set line-length) request, **29**
local motions, 108
\ (unpaddable space) function, **112**
\& (zero-width non-printing) function, **113**
local motions, *continued*
\^ (thin space) function, **112**
\| (thick space) function, **112**
\0 (digit-size space) function, **110**
\b (bracket) function, **117**
\d (move down) function, **107**
\h (horizontal motion) function, **109**
\l (horizontal line) function, **119**
\L (vertical line) function, **120**, 119
\o (overstrike) function, **114**
\r (reverse line) function, **119**
\u (move up) function, **107**
\v (vertical motion) function, **108**
\z (zero motion) function, **115**
long lines, 8
.ls (change line spacing) request, **43**
.lt (set length of title) request, **69**

M

macros, 7, 85
append to, 92
arguments to, 89
copy mode, 92
defining, 85
embedded blanks, 91
invoking, 85
print names and sizes, 133
remove, 87
renaming, 88
margin character, 121
margins on a page
with nroff and troff, 17, 29
mark
horizontal position, 117
vertical position, 37, 94
.mc (margin character) request, **121**
measure, 29
.mk (mark vertical position) request, **37**, 94
mo (month of year) number register, **99**

N

.n (text length) number register, **14**
.na (no adjust) request, **18**
.ne (need space) request, **36**
need space, 36
new page, 35
.nf (no fill) request, **19**
.nh (no hyphenation) request, **21**, 20
nl (vertical position of last baseline) number register, **99**, 93
.nm (number lines) request, **125**
.nn (no number) request, **126**
no adjust request, 18
no fill request, 19
no hyphenation request, 20, 21
no space mode request, 45
no-break control character setting, 124
non-printing character, 113
.nr (set number register) request, **99**
nroff command
exit from, 78
introduction to, 1, 11

.ns (no space mode) request, **45**
number registers, **99**
    access format, 103
    auto-incrementing, 101
    creating, 99
    expressions, 102
    interpolating, 99
    removing, **105**
    setting, 99
numbering lines, **125**, 126
.nx (next file) request, **75**

O

\o (overstrike) function, **114**
.o (page-offset) number register, **29**
one-twelfth em-space, 112
orphans, 37
.os (output saved vertical space) request, **45**
output saved vertical request, 45
overstriking, 114

P

\p (break and spread) function, **15**
.p (page-length) number register, **35**
padding indicators, 62
page length changes, 35
page number, 36, 70
page traps, 94
page-offset, 29
.pc (set page number character) request, **70**
.pi (pipe to program) request, **75**
pipe to program, 75
.pl (set page length) request, **35**
.pm (print macros) request, **133**
.pn (set page number) request, **36**
.po (set page-offset) request, **29**
point size request, 41
predefined number registers
    % — page-number, **36**, 99
    .$ — number of arguments, **89**
    .a — post-line extra space, **44**
    .d — vertical place in current diversion, **94**
    .f — current font, **52**
    .h — text high-water mark, **14**, 94
    .i — current indent, **32**, 34
    .j — current adjustment indicator, **17**
    .l — line-length, **30**
    .n — text length, **14**
    .o — page-offset, **29**
    .p — page-length, **35**
    .s — point-size, **41**
    .t — distance to next trap, 93, 95
    .u — fill mode indicator, **19**
    .v — vertical spacing, **43**
    .z — name of current diversion, **94**
    ct — character type, **117**
    dl — width of last finished diversion, **93**
    dn — height of last finished diversion, 93
    dy — day of month, **99**
    mo — month of year, **99**
    nl — vertical position of last baseline, **99**, 93
predefined number registers, *continued*
    sb — string depth below baseline, **116**
    st — string height above baseline, **116**
    yr — last two digits of year, **99**
print macros, 133
Procrustean mold, 19
.ps (change point size) request, **41**

R

\r (reverse line) function, **119**
.rd (read standard input) request, **76**
read-only number registers
    .$ — number of arguments, **89**
    .a — post-line extra space, **44**
    .d — vertical place in current diversion, **94**
    .f — current font, **52**
    .h — text high-water mark, **14**, 94
    .i — current indent, **32**, 34
    .j — current adjustment indicator, **17**
    .l — line-length, **30**
    .n — text length, **14**
    .o — page-offset, **29**
    .p — page-length, **35**
    .s — point-size, **41**
    .t — distance to next trap, 93, 95
    .u — fill mode indicator, **19**
    .v — vertical spacing, **43**
    .z — name of current diversion, **94**
reading from standard input, 76
referencing strings, 80
removing
    macro definitions, 87
    number registers, 105
    string definitions, 87
renaming macros and strings, 88
requests, 6
    .ab — abort, **134**
    .ad — adjust, **17**
    .af — format of number register, **103**
    .am — append to a macro, **92**
    .as — append to string, **81**
    .bd — break line, **50**
    .bp — begin page, **35**
    .br — break line, **16**, 15
    .c2 — set no-break control character, **124**
    .cc — set control character, **124**
    .ce — center lines, **24**, 23 *thru* 24
    .ch — change position of a trap, **96**
    .cs — constant spacing, **46**
    .cu — continuously underline, **25**
    .da — append to a diversion, **94**
    .de — define macro, **85**
    .di — divert text, **94**
    .ds — define string, **80**
    .dt — set a diversion trap, **96**
    .ec — set escape character, **123**
    .el — else conditional, **129**
    .em — set the end-of-processing trap, **97**
    .eo — set escape off, **123**
    .ev — switch environment, **135**
    .ex — exit from nroff or troff, **78**
    .fc — set field characters, **62**
    .fi — fill, **19**

requests, *continued*
.fl — flush buffer, **134**
.fp — font position, **49**
.ft — set font, **48**
.fz — force font size, **49**
.hc — hyphenation character, **22**
.hw — hyphenate word, **21**
.hy — hyphenate, **20**, 21
.ie — if-else conditional, **129**
.if — conditional processing, **127**
.ig — ignore lines, **130**
.in — indent, **31**
.it — set an input-line-count trap, **96**
.lc — set leader character, **61**
.lg — set ligature mode, **53**
.ll — set line-length, **29**
.ls — line spacing, **43**
.lt — set length of title, **69**
.mc — margin character, **121**
.mk — mark vertical position, **37**, 94
.na — no adjust, **18**
.ne — need space, **36**
.nf — no fill, **19**
.nh — no hyphenation, **21**, 20
.nm — number lines, 125
.nn — no numbering, 126
.nr — set number register, **99**
.ns — no space mode, **45**
.nx — read next source file, **75**
.os — output saved vertical space, **45**
.pc — set page number character, **70**
.pi — pipe to program, **75**
.pl — set page length, **35**
.pm — print macros, **133**
.pn — set page number, **36**
.po — set page-offset, **29**
.ps — point size, **41**
.rd — read from standard input, **76**
removing, 87
renaming, 88
.rm — remove request, macro, or string, **87**
.rn — rename request, macro, or string, **88**
.rr — remove number register, **105**
.rs — restore space mode, **45**
.rt — return to position, **38**, 94
.so — switch source file, **73**
.sp — space, **39**
.ss — set space size, **46**
.sv — save vertical space, **44**
.ta — set tab stops, **55**
.tc — set tab character, **57**
.ti — temporary indent, **32**
.tl — define title, **71**
.tm — terminal message, **78**
.tr — translate characters, **124**
.uf — underline font, **25**
.ul — underline, **24**
.vs — vertical spacing, **43**
.wh — when *something*, **95**, 68
resolution, 8
restore space mode request, 45
return to marked vertical position, 94
return to vertical position, 38
reverse line function, 119
revision bars, 121
right-adjusted tabs, 56
.rm (remove request, macro, or string) request, **87**
.rn (rename request, macro, or string) request, **88**
.rr (remove number register) request, **105**
.rs (restore space mode) request, **45**
.rt (return to position) request, **38**, 94
rules
horizontal, 119
vertical, 119, 120
running headers and footers, 67, 71

S

.s (point-size) number register, **41**
save vertical space request, 44
saving state, 135
sb (string depth below baseline) number register, **116**
sentence endings, 14
set font request, 48
set ligature mode request, 53
set page number, 36
set space-character size request, 46
setting line-length, 29
setting number registers, 99
setting tabs, 55
skipping input lines, 130
.so (switch source) request, **73**
.sp (get vertical space) request, **39**
space request, 39
spaces, 15
.ss (set space-character size) request, **46**
st (string height above baseline) number register, **116**
standard input
reading troff input from, 76
start line numbering, 125
start new page, 35
strings, 79
accessing, 80
appending to, 81
beginning with blanks, 80
defining, 80
removing, 87
renaming, 88
substituting characters, 124
suspend line numbering, 126
.sv (save vertical space) request, **44**
switch source file, 73

T

.t (distance to next trap) number register, **93**, 95
.ta (set tab stops) request, **55**
tabs
absolute, 56
centered, 56
relative, 56
replacement character, 57
right-adjusted, 56
setting, 55

.tc (set tab character) request, **57**
temporary indent of one line, 32
text lines
    as troff input, 6
    ignoring, 130
    words in, 13
thick space, 112
thin space, 112
three-part titles, 71
.ti (temporary indent) request, **32**
title length, 69
titles, 67
.tl (title) request, **71**
.tm (terminal message) request, **78**
.tr (translate characters) request, **124**
translating characters, 124
transparent throughput, 8
traps
    change position of, 96
    diversion, 96
    end-of-processing, 97
    input-line-count, 96
    page, 94
troff command
    exit from, 78
    introduction to, 1, 11
turn escape mechanism on and off, 123

U

\u (move up) function, **107**
.u (fill mode indicator) number register, **19**
.uf (underline font) request, **25**
.ul (underline) request, **24**
underline font request, 25
underline request, 24
units, 8
unpaddable space, 13

V

\v (vertical motion) function, **108**
.v (vertical spacing) number register, **43**
vertical lines, 119, 120
vertical motion, 108
vertical position
    mark, 37
    return to, 38
vertical spacing request, 43
.vs (change vertical spacing) request, **43**

W

\w (width) function, **116**
.wh (when *something*) request, **95**, 68
when *something* request, 68, 95
width function, 116
word, 13

X

\x (get extra line space) function, **44**

Y

yr (last two digits of year) number register, **99**

Z

\z (zero motion) function, **115**
.z (name of current diversion) number register, **94**
zero motion function, 115
zero-width character, 14, 113