Forward

TECHNOLOGY INCORPORATED

XENIX™ SYSTEM

TEXT PROCESSING

VOLUME 3

CONTENTS

1.0 INTRODUCTION.................................... 1-1

2.0 USING THE TEXT EDITORS ED AND SED.................. 2-1
2.1 ED.. 2-4
    2.1.1 A Summary of Commands and Line Numbers 2-4
    2.1.2 More Advanced Editing Techniques 2-6
    2.1.3 Editing Scripts 2-30
2.2 SED... 2-31
    2.2.1 Overall Operation 2-31
    2.2.2 Command-line Flags 2-32
    2.2.3 Order of Application of Editing Commands 2-32
    2.2.4 Pattern-space 2-33
    2.2.5 Addresses 2-33
    2.2.6 Functions 2-34

3.0 PATTERN RECOGNITION AND FILE COMPARISON UTILITIES.......... 3-1
3.1 GREP 3-2
3.2 AWK.. 3-5
3.3 DIFF 3-16
3.4 DIFF3 3-18
3.5 COMM 3-19
3.6 SPELL 3-20

4.0 TEXT FORMATTING AND DOCUMENT PREPARATION......... 4-1
4.1 FORMATTING PACKAGES.............................. 4-2
4.2 SUPPORTING TOOLS................................. 4-3
4.3 HINTS FOR PREPARING DOCUMENTS.................... 4-4
4.4 A NOTE ABOUT THE PAPERS.......................... 4-5
    4.4.1 Using the -ms Macros with Troff and Nroff
    4.4.2 A Guide to Preparing Documents with -ms
    4.4.3 NROFF/TROFF User's Manual
    4.4.4 A TROFF Tutorial
    4.4.5 Tbl- A Program to Format Tables
    4.4.6 Typesetting Mathematics- User's Guide
    4.4.7 Some Applications of Inverted Indexes
5.0 COMMAND REFERENCE................................ 5-1

4.3 HINTS FOR PREPARING DOCUMENTS.................... 4-4
4.4 A NOTE ABOUT THE PAPERS.......................... 4-5
4.4.1 Using the -ms Macros with Troff and Nroff
4.4.2 A Guide to Preparing Documents with -ms
4.4.3 NROFF/TROFF User's Manual
4.4.4 A TROFF Tutorial
4.4.5 Tbl- A Program to Format Tables
4.4.6 Typesetting Mathematics- User's Guide
4.4.7 Some Applications of Inverted Indexes

CHAPTER 1

INTRODUCTION

Users involved in text processing applications like typing memos, writing technical reports, and preparing documentation, will soon discover that their primary interface with the computer is through the editors, the various pattern recognition and file comparison utilities, and the text formatting packages. Programmers also make extensive use of the editors and other utilities described in this volume for writing and revising code. Therefore, it is extremely important that all users learn as much as possible about the tools available to them on the XENIX system, and practice using the various commands and functions. The more understanding the user has of which functions work best in which situations and the more dexterity the user developes in using particular commands, the more powerful the editors and related tools become.

This volume contains an introduction to the XENIX text editors, **ed** and **sed**. For a more detailed tutorial material concerning the XENIX text editors, read the appropriate sections in The Programmer's Introduction.

Also introduced in this volume are some tools which prove extremely useful in the process of preparing documents, when it is necessary to locate repeated elements in a single file or group of files to make a consistent set of changes, or to compare and contrast two or more files in order to identify the differences between them. Because several of these programs may be used interchangeably, knowing which one will do the job at hand most efficiently is a large part of understanding their use. These programs streamline complicated editing command procedures, locate variations among several versions of text, and can deal with large numbers of text files at once.

- members of the **grep** family, **grep**, **egrep**, and **fgrep**.
- **awk**.
- **diff** and **diff3**.
- **comm**.
- **spell**.

The XENIX system also offers two text formatting packages which simplify the production of technical reports, memoranda, formal papers, and documentation, **nroff** and **troff** designed to produce output for the lineprinter and typesetter, respectively. **-Ms**, a canned package of formatting requests which is much simpler to use than **nroff** and **troff**, is described in detail. Some supporting programs that aid in document preparation, including **eqn** which integrates mathematical symbols and equations into the text of a document, **tbl** which provides an analogous service for preparing tabular material, and, **refer** which prepares bibliographic citations from a data base, are also discussed in this volume.

CHAPTER 2

USING THE TEXT EDITORS ED AND SED

Most users of a computer system rely heavily on text editors in doing their work, whether it be writing programs or preparing data. For those users involved in text processing applications, for typing memos, writing technical reports, or preparing documentation, the various editing functions may be their primary interface with the computer. Therefore, it is extremely important that the text processing user learn as much as possible about the editing tools available on the system, and practice using the various commands and functions. The more understanding the user has of which functions work best in which situations and the more dexterity the user developes in using particular commands, the more powerful the editors and related tools become. For a more detailed introduction to text editing with XENIX, read the appropriate sections in _The Programmer's Introduction_.

XENIX offers two text editors, **ed**, an interactive line editor, and **sed**, a non-interactive context editor. Although in many respects the capabilities of these two editors overlap, the user will soon find that **ed** is more appropriate to on-the-spot entry, deletion and simple modification of text. Sed is more appropriate when uniform changes must be made in large files or groups of file, or when the sequence of editing commands needed to make the changes becomes complex.

Because **sed** is derived from **ed**, however, the two editors share some characteristics. In particular, they recognize the same class of _regular expressions_. A regular expression specifies a set of strings of characters to be matched by a pattern found in the text, sometimes referred to as a context address. In practical terms, these are the patterns the user asks the editor to search and substitute when changes in text are required. These regular expressions include:

1. An ordinary character (not one of those discussed below) is a regular expression, and matches that character.

2. A circumflex \`^' at the beginning of a regular expression matches the null character at the beginning of a line.

3. A dollar-sign `$' at the end of a regular expression matches the null character at the end of a line.

4. The characters `\n' match an imbedded newline character, but not the newline at the end of the pattern space.

5. A period `.' matches any character except the terminal newline of the pattern space.

6. A regular expression followed by an asterisk `*' matches any number (including 0) of adjacent occurrences of the regular expression it follows.

7. A string of characters in square brackets `[ ]' matches any character in the string, and no others. If, however, the first character of the string is circumflex `^', the regular expression matches any character <u>except</u> the characters in the string and the terminal newline of the pattern space.

8. A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.

9. A regular expression between the sequences `\(' and `\)' is identical in effect to the unadorned regular expression, but has side-effects which are described under the <u>s</u> command below and specification 10) immediately below.

10. The expression `<u>\d</u>' means the same string of characters matched by an expression enclosed in `\(' and `\)' earlier in the same pattern. Here <u>d</u> is a single digit; the string specified is that beginning with the <u>d</u>th occurrence of `\(' counting from the left. For example, the expression `^\(.*\)\1' matches a line beginning with two repeated occurrences of the same string.

11. The null regular expression standing alone (e.g., `//') is equivalent to the last regular expression compiled.

To use one of the special characters (^ $. * [ ] \ /) as a literal (to match an occurrence of itself in the input), precede the special character by a backslash `\'.

For a context address to `match' the input requires that the whole pattern within the address match some portion of the pattern space. The use of these pattern matches for

specific applications within **ed** and sed are discussed in detail for each editor.

2.1 ED

Ed is one of the text editors on the XENIX system, used primarily to create and modify text interactively, whether it is a document, a program, or data for a program. The most frequently used commands are summarized here, followed by a discussion of editing techniques especially useful in text processing applications.

2.1.1 A Summary of Commands and Line Numbers

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

a Append, that is, add lines to the buffer (at line dot, unless a different line is specified). Appending continues until a period is is typed on a new line. The value of dot is set to the last line appended.

c Change the specified lines to the new text which follows. The new lines are terminated by a period on a newline, as with **a**. If no lines are specified, replace line dot. Dot is set to last line changed.

d Delete the lines specified. If none are specified, delete line dot. Dot is set to the first undeleted line, unless $ is deleted, in which case dot is set to $.

e Edit new file. Any previous contents of the buffer are thrown away, so issue a **w** beforehand.

f Print remembered filename. If a name follows **f**, then the remembered name is set to it.

g The command

```
g/---/commands
```

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

i Insert lines before specified line (or dot) until a single period is typed on a new line. Dot is set to the last line inserted.

m Move lines specified to after the line named after **m**. Dot is set to the last line moved.

p Print specified lines. If none are specified, print line the line specified by dot. A single line number is equivalent to the line-numberp command. A single <RETURN> prints .+1, the next line.

q Quit **ed**. Wipes out all text in buffer if given twice in a row without a **w** command.

r Read a file into buffer (at end unless specified elsewhere.) Dot is set to the last line read.

s The command

    s/string1/string2/

substitutes the characters string1 into string2 in the specified lines. If no lines are specified, the substitution takes place only on the line specified by dot. Dot is set to the last line in which a substitution took place, which means that if no substitution takes place, dot remains unchanged s changes only the first occurrence of string1 on a line; to change all of them, type a **g** after the final slash.

v The command

    v/---/commands

executes commands on those lines that do not contain ---.

w Write out buffer onto a file. Dot remains unchanged.

.= Print value of dot. (Anequalsignbyitself prints the value of $.)

! The line

    !command-line

causes command-line to be executed as a XENIX command.

/string/ Context search. Search for next line which contains this string of characters. Print it. Dot is set to the line where string was found. Search starts at .+1 , wraps around from $ to 1, and continues to dot, if necessary.

?string? Context search in reverse direction. Start search at .-1 , scan to 1, wrap around to $.

2.1.2 More Advanced Editing Techniques

There are often several alternative procedures for accomplishing the same editing task, with varying degrees of efficiency. This section provides explanations and examples of how to use ed to edit with less effort and greater speed.

Topics covered include

- Special characters in search and substitute commands
- Line addressing
- Global commands
- Line moving
- Line copying

2.1.2.1 Special Characters

There are several special characters which facilitate searching and substitution in ed.

The List command `l' **Ed** provides two commands for printing the contents of lines. One of these is p, in combinations like

```
1,$p
```

to print all the lines in the file, or

```
s/abc/def/p
```

to change `abc' to `def' on the current line. Less familiar is the list command l (the letter `l'), which gives slightly more information than p. In particular, l makes visible characters that are normally invisible, such as tabs and backspaces. l prints each tab as > and each backspace as <, in a line which contains these characters. This makes it much easier to correct typing mistakes when extra spaces are adjacent to tabs, or backspaces are followed by a space.

The l command also `folds' long lines, by printing any line that exceeds 72 characters on multiple lines; each printed line except the last is terminated by a backslash \, to

indicate that it was folded. This overcomes the limitation of your terminal screen width.

Occasionally, the **l** command will print in a line a string of numbers preceded by a backslash, such as \07 or \16, making visible characters that normally do not print, like form feed or vertical tab or bell, usually typed in error. These combinations are a single character with special meanings on some terminals.

The Substitute Command `s' The substitute command **s** is the command for changing the contents of individual lines, probably the most complicated and powerful of ed commands. The most straightforward example is the trailing **g** after a substitute command. With

```
s/this/that/
```

and

```
s/this/that/g
```

the first one replaces the first `this' on the line with `that'. If there is more than one `this' on the line, the second form with the trailing **g** changes all of them.

Either form of the **s** command can be followed by **p** or **l** to `print' or `list' the contents of the line:

```
s/this/that/p
s/this/that/l
s/this/that/gp
s/this/that/gl
```

are all slight variations.

Of course, any **s** command can be preceded by one or two `line numbers' to specify that the substitution is to take place on a group of lines. Thus

```
1,$s/mispell/misspell/
```

changes the first occurrence of `mispell' to `misspell' in each line of the file. But

```
1,$s/mispell/misspell/g
```

changes every occurrence in each line.

If a **p** or **l** is added to the end of any of these substitute commands, only the last line that got changed will be printed, not all the lines.

The Undo Command `u'  If a substitution in a line is incorrect, The `undo' command u will `undo' the last substitution: the last line that was substituted can be restored to its previous state by typing the command

```
u
```

The Metacharacter `.'  Certain characters have special meanings when they occur in the left side of a substitute command, or in a search for a particular line. These special characters, are called `metacharacters.'

The first one is the period `.'. On the left side of a substitute command, or in a search with /.../, `.' stands for any single character. Thus the search

```
/x.y/
```

finds any line where `x' and `y' occur separated by a single character, as in

```
x+y
x-y
x y
xzy
```

and so on.

Since `.' matches any single character, it can be used to delete or substitute special characters printed by **l**. If a line printed with the **l** command, appears as

```
th\07is
```

which represents the bell character,

```
s/\07//
```

will not delete it. Retyping the entire line is inefficient. The metacharacter `.', used in the following way

```
s/th.is/this/
```

will do the job. The `.' matches whatever character is between the `h' and the `i'.

Since `.' matches any character, the command

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

converts the first character on a line into a comma `,'; it does not correct the punctuation.

Like other special characters in **ed**, the `.' has several meanings, depending on its context. This line demonstrates three:

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

The first period is the current line number, which is called `dot'. The second `.' is a metacharacter that matches any single character on that line. The third `.' is the literal period. On the right side of a substitution, `.' is not special.

The Backslash `\' Sometimes a period is really required. To convert the line

```
Now is the time.
```

into

```
Now is the time?
```

use the backslash `\'. A backslash turns off any special meaning that the next character might have; in particular, `\.' converts the `.' from a `match anything' into a period, so It can replace the period in

```
Now is the time.
```

like this:

```
s/\./?/
```

The pair of characters `\.' is considered by **ed** to be a single real period.

The backslash can also be used when searching for lines that contain a special character. The following example looks for a line that contains

```
.DE
```
at the start of a line.

The search

```
/.DE/
```

isn't adequate, for it will find lines like

```
JADE
FADE
MADE
```

because the `.' matches each of the letters `A'. But the search

```
/\.DE/
x...
```

will result in lines that contain `.DE'.

The backslash can also be used to turn off special meanings for characters other than `.'. For example, consider finding a line that contains a backslash. The search

```
/\/
```

won't work, because the `\' isn't a literal backslash, but instead means that the second `/' no longer delimits the search. By preceding a backslash with another one, a literal backslash can be searched. Thus /\\/ does work. Search for a forward slash `/' with

```
/\//
```

The backslash turns off the special meaning of the immediately following `/' so that it doesn't terminate the /.../ construction prematurely.

As an exercise, find two substitute commands each of which will convert the line

```
\x\.\y
```

into the line

```
\x\y
```

Here are several solutions; verify that each works as advertised.

```
s/\\\.//
s/x../x/
s/..y/y/
```

Any character can delimit the pieces of an **s** command: there is nothing sacred about slashes. (slashes are necessary for context searching.) For instance, in a line that contains a lot of slashes already, like

```
//exec //sys.fort.go // etc...
```

a colon can be used as the delimiter; to delete all the slashes, type

```
s:/::g
```

Second, if `#' and `@' are the character erase and line kill characters, it is necessary to type \# and \@, whether in ed or any other program.

When adding text with **a** or **i** or **c**, backslash is not special; use only the backslashes actually required.

The Circumflex `^' The circumflex `^' stands for the beginning of the line. For example, searching for a line that begins with `the', the command

```
/the/
```

will probably locate several lines that contain `the' in any position. With

```
/^the/
```

the context is specified.

The other use of `^' is to insert something at the beginning of a line:

```
s/^/ /
```

places a space at the beginning of the current line.

Metacharacters can be combined. To search for a line that contains only the characters

```
.PP
```

the command

```
/^\.PP$/
```

can be used.

The Star `*' It is possible to replace all the spaces between **x** and **y** with a single space in a line that looks like this:

```
text x                    y text
```

where **text** stands for lots of text, and there are some indeterminate number of spaces between the **x** and the **y** without retyping the whole line.

The metacharacter `*' following any character stands for any number of consecutive occurrences of that character. To refer to all the spaces at once, use

```
s/x *y/x y/
```

The construction ` *' means `as many spaces as possible'. Thus `x *y' means `an x, as many spaces as possible, then a y'.

The star can be used with any other character. If the original example was instead

```
text x--------y text
```

then all `-' signs can be replaced by a single space with the command

```
s/x-*y/x y/
```

To change

```
Now is the time for all good men ....
```

into

```
Now is the time.
```

use `.*' to eat up everything after the `for':

```
s/ for.*/./
```

The star `*' should be used very carefully. Note especially that `as many as possible' occurrences can mean zero. if a line contains

```
xy text x  y text
```

and we said

```
s/x *y/x y/
```

the <u>first</u> `xy' matches this pattern, for it consists of an `x', zero spaces, and a `y'. The result is that the substitute acts on the first `xy', and does not touch the later one that actually contains some intervening spaces.

To avoid this, specify a pattern like

```
/x  *y/
```

which creates `an x, a space, then as many more spaces as possible, then a y', in other words, one or more spaces.

The other startling behavior of `*' is again related to the fact that zero is a legitimate number of occurrences of something followed by a star. The command

```
s/x*/y/g
```

when applied to the line

```
abcdef
```

produces

```
yaybycydyeyfy
```

which is almost certainly not what was intended. The reason for this behavior is that zero is a legal number of matches, and there are no x's at the beginning of the line (so that gets converted into a `y'), nor between the `a' and the `b' (so that gets converted into a `y'), and so on. To avoid zero matches, write

```
s/xx*/y/g
```

`xx*' is one or more x's.

<u>Brackets `[ ]'</u> In deleting any numbers that appear at the beginning of all lines of a file, it is possible to use a series of commands like

```
1,$s/^1*//
1,$s/^2*//
1,$s/^3*//
```

and so on, but this is a lengthy process if the numbers are at all long. To remove all the digits on one pass, use the brackets [and].

The construction

```
[0123456789]
```

matches any single digit -- the whole thing is called a `character class'. The pattern `[0123456789]*' matches zero or more digits (an entire number), so

```
1,$s/^[0123456789]*//
```

deletes all digits from the beginning of all lines.

Any characters appears within a character class, with no special characters inside the brackets; even the backslash doesn't have a special meaning. To search for special characters, for example, try

```
/[.\$^[]/
```

Within [...], the `[' is not special. To get a `]' into a character class, make it the first character.

The digits can be abbreviated as [0-9]; similarly, [a-z] stands for the lower case letters, and [A-Z] for upper case.

Also a class that means `none of the following characters'. can be specified using a `^':

```
[^0-9]
```

stands for `any character <u>except</u> a digit'. To find a first line that doesn't begin with a tab or space use:

```
/^[^(space)(tab)]/
```

Within a character class, the circumflex has a special meaning only if it occurs at the beginning.

<u>The Ampersand `&'</u> The ampersand `&' is used primarily to save typing. To change

```
Now is the time
```

to

```
Now is the best time
```

without repeating the word `the'; the ampersand `&' can be used. On the <u>right</u> side of a substitute, the ampersand means `whatever was just matched'. In

```
s/the/& best/
```

the `&' will stand for `the'. This can save typing and potential errors in complicated text. For example, to parenthesize a line, regardless of its length, type:

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

The ampersand can occur more than once on the right side:

```
s/the/& best and & worst/
```

makes

```
Now is the best and the worst time
```

and

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

converts the original line into

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

To get a literal ampersand, the backslash is used to turn off the special meaning. For example,

```
s/ampersand/\&/
```

converts the word into the symbol. Notice that `&' is not special on the left side of a substitute, only on the right side.

Substituting Newlines **Ed** provides a facility for splitting a single line into two or more shorter lines by substituting in a newline. As the simplest example, suppose a line has gotten unmanageably long because of editing. If it looks like

```
text    xy   text
```

it can be broken between the `x' and the `y' like this:

```
s/xy/x\
y/
```

This is actually a single command, although it is typed on two lines. Since `\' turns off special meanings, a `\' at the end of a line makes the newline there no longer special.

A single line can be made into several lines with this same mechanism. For example, consider underlining the word `very' in a long line by splitting `very' onto a separate line, and preceding it by the formatting command `.I':

text a very big text

The command

```
s/ very /\
.I\
very\
/
```

converts the line into four shorter lines, preceding the word `very' by the line `.I', and eliminating the spaces around the `very', all at the same time.

When a newline is substituted in a string, dot is left pointing at the last line created.

Joining Lines Lines may also be joined together, but this is done with the j command instead of s. Given the lines

```
Now is
 the time
```

and supposing that dot is set to the first of them, then the command

```
j
```

joins them together. No blanks are added, because a blank was included at the beginning of the second line.

All by itself, a j command joins the lines signified by dot and dot + 1, but any contiguous set of lines can be joined. Just specify the starting and ending line numbers. For example,

```
1,$jp
```

joins all the lines into one big one and prints it.

Rearranging a Line with \(... \) `&' stands for whatever is matched by the left side of an **s** command. It is possible to specify which parts of a line need to be matched in order to rearrange them. If a file consists of names in the form

```
Smith, A. B.
Jones, C.
```

and so on, to get the initials to precede the name, as in

```
A. B. Smith
C. Jones
```

`tag' the pieces of the pattern (in this case, the last name, and the initials), and then rearrange the pieces. On the left side of a substitution, if part of the pattern is enclosed between \(and \), whatever matched that part is remembered, and available for use on the right side. On the right side, the symbol `\1' refers to whatever matched the first \(...\) pair, `\2' to the second \(...\), and so on.

The command

```
1,$s/^\([^,]*\), *\(.*\)/\2 \1/
```

although hard to read, does the job. The first \(...\) matches the last name, which is any string up to the comma; this is referred to on the right side with `\1'. The second \(...\) is whatever follows the comma and any spaces, and is referred to as `\2'.

Any editing sequence this complicated should be used with the global commands **g** and **v** to print exactly those lines which were affected by the substitute command, and verify that each case is correct.

2.1.2.2 Line Addressing in the Editor It is important to understand how line addressing works in ed, in order to be able to specify which lines will be affected by editing commands. Constructions like

```
1,$s/x/y/
```

specify a change on all lines. Using a single newline or return to print the next line, with

```
/thing/
```

will find a line that contains `thing'.

```
?thing?
```

can be used to scan backwards for the previous occurrence of `thing', a useful feature if the item is above the current line. The slash and question mark are the only characters which delimit a context search, although essentially any character can be used in a substitute command.

Address Arithmetic The next step is to combine the line numbers like `.', `$', `/.../' and `?...?' with `+' and `-'. Thus

```
$-1
```

is a command to print the next to last line of the current file (that is, one line before line `$'). For example,

```
$-5,$p
```

prints the last six six lines of a file.

As another example,

```
.-3,.+3p
```

prints from three lines before the current line to three lines after, to provide some context. The `+' can be omitted:

```
.-3,.3p
```

is identical in meaning.

Using `-' and `+' as line numbers by themselves saves typing effort. Move back up one line in the file for each minus sign:

```
---
```

moves up three lines, as does `-3'.

Since `-' is shorter than `.-1', constructions like

```
-,.s/bad/good/
```

are useful. This changes `bad' to `good' on the previous line and on the current line.

`+' and `-' can be used in combination with searches using `/.../' and `?...?', and with `$'. The search

```
/thing/--
```

finds the line containing `thing', and positions the current line two lines before it.

<u>Repeated Searches</u> Suppose the search

```
/horrible thing/
```

finds the wrong horrible thing, and it is necessary to repeat the search. Instead of retyping the search, use

```
//
```

as shorthand for `the previous thing that was searched for'. This can be repeated as many times as necessary. In addition

```
??
```

searches for the same thing, but in the reverse direction.

`//' can be used as the left side of a substitute command, to mean `the most recent pattern'.

```
/horrible thing/
 ... ed prints line with `horrible thing' ...
s//good/p
```

To go backwards and change a line, use

```
??s//good/
```

Of course, the `&' on the right hand side of a substitute can still be used to stand for whatever got matched:

```
//s//& &/p
```

finds the next occurrence of whatever was searched last, replaces it by two copies of itself, then prints the line just to verify that it worked.

<u>Default Line Numbers and the Value of Dot</u> To speed up editing know what lines will be affected by a command if no lines are specified, and what the current line will be when a command finishes.

A search command like

```
/thing/
```

makes the current line the next line that contains `thing'. Then no address is required with commands like **s** to make a substitution on that line, or **p** to print it, or **l** to list it, or **d** to delete it, or **a** to append text after it, or **c** to change it, or **i** to insert text before it.

If the search was unsuccessful the position remains unchanged. This is also true if the current position was the only occurence of thing'. The same rules hold for searches that use `?...?'.

The delete command **d** leaves dot pointing at the line that followed the last deleted line. When line `$' gets deleted, however, dot points at the new line `$'.

The line-changing commands **a**, **c** and **i** by default all affect the current line; if no line number is specified, a appends text after the current line, **c** changes the current line, and **i** inserts text before the current line.

a, **c**, and **i** point at the last line entered when appending, changing or inserting is over. For example,

```
a
text
botch   (minor error)
.
s/botch/correct/  (fix botched line)
a
more text
.
```

works without specifying any line number for the substitute command or the second append command, as will

```
a
text
horrible botch  (major error)
.
c               (replace entire line)
fixed up line
```

The **r** command will read a file into the text being edited, either at the end if no address is given, or after the specified line. In either case, dot points at the last line read in. Remember that

```
0r
```

will read a file in at the beginning of the text. 0a or 1i can be used to start adding text at the beginning.

The **w** command writes out the entire file. one line number preceding a command, writes that line. A command preceded by two line numbers, writes that range of lines. The **w** command does not change dot: the current line remains the same, regardless of what lines are written.

Since the **w** command is so easy to use, it should be used periodically while editing in case of system crashes or disasterous errors.

With the **s** command, the rule is that the current position is the last line changed; if there are no changes, then the position is unchanged.

To illustrate, suppose that there are three lines in the buffer, and the line given by dot is the middle one:

```
x1
x2
x3
```

Then the command

```
-,+s/x/y/p
```

prints the third line, which is the last one changed. But if the three lines had been

```
x1
y2
y3
```

and the same command had been issued while dot pointed at the second line, then the result would be to change and print only the first line, and that is where dot would be set.

Semicolon `;' Searches with `/.../' and `?...?' start at the current line and move forward or backward respectively until they either find the pattern or get back to the current line. If this is not what is wanted, as in this example:

```
.
.
.
ab
.
.
.
bc
.
.
```

Starting at line 1,

```
/a/,/b/p
```

does not print all the lines from the `ab' to the `bc' inclusive. Both searches (for `a' and for `b') start from the same point, and both find the line that contains `ab'. The result is a single line. Worse, if there had been a line with a `b' in it before the `ab' line, then the print command would be in error, since the second line number would be less than the first, and it is illegal to try to print lines in reverse order. This is because the comma separator for line numbers doesn't set dot as each address is processed; each search starts from the same place.

In ed, the semicolon `;' can be used just like comma, with the single difference that use of a semicolon forces dot to be set at that point as the line numbers are being evaluated. In effect, the semicolon `moves' dot. Thus in the example above, the command

```
/a/;/b/p
```

prints the range of lines from `ab' to `bc', because after the `a' is found, dot is set to that line, and then `b' is searched for, starting beyond that line.

This property is useful if searching for the second occurrence of `thing'. Instead of

```
/thing/
//
```

which prints the first occurrence as well as the second, use

```
/thing/;//
```

This finds the first occurrence of `thing', sets dot to that line, then finds the second and prints only that.

Closely related is searching for the second previous occurrence of something, as in

```
?something?;??
```

To find the first occurrence of something in a file, starting at an arbitrary place within the file, it is not sufficient to use

```
1;/thing/
```

because this fails if `thing' occurs on line 1. But it is possible to use

```
0;/thing/
```

for this starts the search at line 1.

Interrupting the Editor If the interrupt, delete, rubout, or break keys are used while **ed** is executing a command, the file is restored as much as possible to what it was before the command began. Naturally, some changes are irrevocable; an interrupted read or write to a file or substitutions or deletions will be stopped in some unpredictable state in the middle. Dot may or may not be changed. Printing is more clear cut. Dot is not changed until the printing is done.

2.1.2.3 Global Commands The global commands g and **v** are used to perform one or more editing commands on all lines that either contain (g) or don't contain (v) a specified pattern.

As the simplest example, the command

```
g/XENIX/p
```

prints all lines that contain the word `XENIX'. The pattern that goes between the slashes can be anything that could be used in a line search or in a substitute command; exactly the same rules and limitations apply.

As another example, then,

```
g/^\./p
```

prints all the formatting commands in a file (lines that begin with `.').

The **v** command is identical to **g**, except that it operates on those line that do not contain an occurrence of the pattern. Mnemonically, the `v' can be thought of as part of the word inverse.

For example

```
v/^\./p
```

prints all the lines that don't begin with `.' -- the actual text lines.

The command that follows **g** or **v** can be anything. For example, the following command deletes all lines that begin with `.':

```
g/^\./d
```

This command deletes all empty lines:

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

Probably the most useful command that can follow a global is the substitute command; it can be make a change and print each affected line for verification. For example, the word `Xenix' could be changed to `XENIX' globally, and simultaneouly verified with

```
g/Xenix/s//XENIX/gp
```

Notice that `//' in the substitute command means `the previous pattern', in this case, `Xenix'. The **p** command is done on each line that matches the pattern, not just those on which a substitution took place.

The global command operates by making two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each marked line is examined, in turn, and dot is set to that line, and the command executed. This means that it is possible for the command that follows a **g** or **v** to use addresses, set dot, and so on, quite freely.

```
g/^\.P/+
```

prints the line that follows each `.P' command. Remember that `+' means `one line past dot' and

```
g/topic/?^\.H?1
```

searches for each line that contains `topic', scans

backwards until it finds a line that begins `.H' (a heading) and prints the line that follows that, thus showing the headings under which `topic' is mentioned. Finally,

```
g/^\.EQ/+,/^\.EN/-p
```

prints all the lines that lie between lines beginning with `.EQ' and `.EN' formatting commands.

The **g** and **v** commands can also be preceded by line numbers, in which case the lines searched are only those in the range specified.

Multi-line Global Commands It is possible to do more than one command under the control of a global command, although the syntax may be awkward. As an example, suppose the task is to change `x' to `y' and `a' to `b' on all lines that contain `thing'. Then

```
g/thing/s/x/y/\
s/a/b/
```

is sufficient. The `\' signals the **g** command that the set of commands continues on the next line; it terminates on the first line that does not end with `\'. A substitute command cannot be used to insert a newline within a **g** command.

The command

```
g/x/s//y/\
s/a/b/
```

does not work as expected. The remembered pattern is the last pattern actually executed, so sometimes it will be `x' (as expected), and sometimes it will be `a' (not expected). It must be spelled out, like this:

```
g/x/s/x/y/\
s/a/b/
```

It is also possible to execute **a**, **c** and **i** commands under a global command; as with other multiline constructions, all that is needed is to add a `\' at the end of each line except the last. Thus, to add a `.nf' and `.sp' command before each `.EQ' line, type:

```
g/^\.EQ/i\
.nf\
.sp
```

There is no need for a final line containing a `.' to terminate the **i** command, unless there are further commands being done under the global.

2.1.2.4 Cut and Paste with the Editor Sometimes the best approach to doing cut and paste work with files is to use the familiar XENIX commands for copying, changing file names, removing files, and putting two or more files together. Often, however, it is necessary to manipulate parts of files, individual lines, or groups of lines. Ed offer several techniques for doing this work conveniently.

Filenames **r** and **w** are the two essential commands for `read' and `write.' There is also the `edit' command e. Within ed, the command

```
e  newfile
```

allows the user to change current working files. It has the same effect as using the **q** command, then reentering ed with a new file name, except that if a pattern is remembered, then a command like // will still work.

The command

```
ed  file
```

remembers the name of the file, and any subsequent e, r or **w** commands that don't contain a filename will refer to this remembered file. Thus,

```
ed  file1
        (editing)
w       (writes back in file1)
e  file2      (edit new file, without leaving editor)
        (editing on file2)
w       (writes back on file2)
```

(and so on) does a series of edits on various files without ever leaving ed and without typing the name of any file more than once. This is extremely useful for making changes in several files at once.

The **f** command without a file name will give the name of the remembered file; the name of the remembered file name can also be changed with **f**; a useful sequence is

```
ed precious
f junk
(editing)
```

which gets a copy of a precious file, then uses **f** to guarantee that a careless **w** command won't clobber the original.

Inserting One File into Another To insert the file called `table' in a file called `memo' just after the reference to Table 1, edit `memo', find `Table 1', and add the file `table' right there:

```
ed memo
/Table 1/
Table 1 shows that ... [response from ed]
.r table
```

The critical line is the last one. The command asks for the file to be read in right after line dot. An **r** command without any address adds lines at the end, so it is the same as $r.

Writing out Part of a File It is also possible to write out part of the file being edited. For example, to split out into a separate file the table from the previous example, so it can be formatted and tested separately:

```
.TS
[lots of stuff]
.TE
```

which is the way a table is set up for the **tbl** program. To isolate the table in a separate file called `table', first find the start of the table (the `.TS' line), then write out that part:

```
/^\.TS/
.TS [ed prints the line it found]
.,/^\.TE/w table
```

or it can be done all at once with

```
/^\.TS/;/^\.TE/w table
```

The **w** command can write out a group of lines, instead of the whole file, or even a single line.

Moving Lines Around Sometimes it is useful to move a paragraph from its present position in a paper to the end. One possibility is to write the paragraph onto a temporary file, delete it from its current position, then read in the temporary file at the end. Assuming that the current line is the `.P' command that begins the paragraph, this is the sequence of commands:

```
.,/^\.P/-w temp
.,//-d
$r temp
```

That is, from the current line (`.') until one line before the next `.P' (`/^\.P/-') write onto `temp'. Then delete the same lines. Finally, read `temp' at the end.

An easier way to do this is to use the move command **m** that ed provides to perform the whole set of operations at once, without any temporary file.

The **m** command is like many other ed commands in that it takes up to two line numbers in front that tell what lines are to be affected. It is also followed by a line number that tells where the lines are to go. Thus

```
line1, line2 m line3
```

moves all the lines between line1 and line2 after line3. Naturally, any of line1, etc., can be patterns between slashes, $ signs, or other ways to specify lines.

If the current line is the first line of the paragraph, then

```
.,/^\.P/-m$
```

will append the paragraph to the end of the file.

Another frequently used operation is reversing the order of two adjacent lines by moving the first one to after the second. Positioned at the first of the two lines, use

```
m+
```

It moves line dot to after one line after line dot. Positioned on the second line,

```
m--
```

does the interchange.

The **m** command is more efficient than writing, deleting and re-reading.

Marks Ed provides a facility for marking a line with a particular name, so that it can be referenced by name, regardless of its actual line number. This can be handy for keeping track of lines as they move. The mark command is k; the command

```
kx
```

marks the current line with the name `x'. If a line number precedes the **k**, that line is marked. (The mark name must be a single lower case letter.) The marked line has the address

```
'x
```

Marks are most useful for moving things around. Find the first line of the block to be moved, and mark it with:

```
'a.
```

Then find the last line and mark it with

```
'b.
```

Positioned at the place where the lines are to be inserted use

```
'a,'bm.
```

Only one line can have a particular mark name associated with it at any given time.

Copying Lines **ed** provides a command called **t** (for `transfer') for making a copy of a group of one or more lines at any point. This is often easier than writing and reading.

The **t** command is identical to the **m** command, except that instead of moving lines it simply duplicates them at the place named. Thus

```
1,$t$
```

duplicates the entire contents being edited. A more common use for **t** is to create a series of lines that differ only slightly. For example:

```
a
.......... x ......... (long line)
.
t.          (make a copy)
s/x/y/      (change it a bit)
t.          (make third copy)
s/y/z/      (change it a bit)
```

The Temporary Escape `!' Sometimes it is convenient to be able to temporarily escape from the editor to do some other XENIX command without leaving the editor. The `escape' command `!' provides a way to do this.

    !command

suspends the current editing state, and the XENIX command is executed. When the command finishes, ed prints another !; at that point editing can be resumed. any XENIX command can be executed in this way, including another **ed**.

2.1.3 Editing Scripts

Ed they can also be used to accomplish a complicated set of editing operations on a group of files using a `script' (i.e. a file that contains a sequence of operations can be written and applied to each file in turn.) Editing scripts can often be used as an alternative to the programs introduced in the next chapter.

For example, to change every `Xenix' to `XENIX' and every `Unix' to `UNIX' in a large number of files, put into the file script the lines

```
g/Xenix/s//XENIX/g
g/Unix/s//UNIX/g
w
q
```

Now use the commands

```
ed file1 <script
ed file2 <script
...
```

This causes **ed** to take its commands from the prepared script. By using the XENIX shell command interpreter, a set of files can be cycled through automatically, with varying degrees of ease, if the job is planned in advance.

2.2 USING SED

Sed is a non-interactive context editor designed to be especially useful in three cases:

1. To edit files too large for comfortable interactive editing.

2. To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode.

3. To perform multiple `global' editing functions efficiently in one pass through the input.

Since only a few lines of the input reside in core at one time, and no temporary files are used, the effective size of file that can be edited is limited only by the requirement that the input and output fit simultaneously into available secondary storage. Complicated editing scripts can be created separately and given to **sed** as a command file. For complex edits, this saves considerable typing, and its attendant errors. **Sed** running from a command file is much more efficient than an interactive editor like ed, even if driven by a pre-written script.

On the other hand, **sed** lacks relative addressing due to line-at-a-time operation, and the user gets no immediate verification that the command has altered the text in the way the user intended.

Although **sed** is a lineal descendant of **ed**, considerable changes have been made between **ed** and **sed**, because of the differences between interactive and non-interactive operation. The most striking family resemblance between the two editors is in the class of regular expressions they recognize; the code for matching patterns is copied almost verbatim from the code for **ed**.

2.2.1 Overall Operation

Sed by default copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. This behavior may be modified by flags on the command line;

The general format of an editing command is:

```
[address1,address2][function][arguments]
```

One or both addresses may be omitted; the format of addresses is given below. Any number of blanks or tabs may separate the addresses from the function, which must be present. the available commands are discussed below. The arguments may be required or optional, according to which function is given, as discussed for each individual function. Tab characters and spaces at the beginning of lines are ignored.

2.2.2 Command-line Flags

Three flags are recognized on the command line:

-n: tells **sed** not to copy all lines, but only those specified by p functions or p flags after s functions;

-e: tells **sed** to take the next argument as an editing command;

-f: tells Sed to take the next argument as a file name; the file should contain editing commands, one to a line.

2.2.3 Order of Application of Editing Commands

Before any input file is opened and editing begins, all the editing commands are compiled into a form which will be efficient during the execution phase, when the commands are actually applied to lines of the input file. The commands are compiled in the order in which they are encountered; this is generally the order in which they will be attempted at execution time. The commands are applied one at a time; the input to each command is the output of all preceding commands.

The default linear order of application of editing commands can be changed by the flow-of-control commands, t and b. Even when the order of application is changed by these commands, it is still true that the input line to any command is the output of any previously applied command.

2.2.4 Pattern-space

The range of pattern matches is called the pattern space. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the N command.

Examples Examples are scattered throughout the text. Except where otherwise noted, the examples all assume the following input text:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

as in this example. The command

```
2q
```

will quit after copying the first two lines of the input. The output will be:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

2.2.5 Addresses

The following rules apply to addressing in sed.

2.2.5.1 Selecting lines for Addressing Lines in the input file(s) to which editing commands are to be applied can be selected by addresses, which may be either line numbers or context addresses. The application of a group of commands can be controlled by one address (or address-pair) by grouping the commands with curly braces (`{ }').

2.2.5.2 Line-number Addresses A line number is a decimal integer. As each line is read from the input, a line-number counter is incremented; a line-number address matches (selects) the input line which causes the internal counter to equal the address line-number. The counter runs cumulatively through multiple input files; it is not reset when a new input file is opened. A special case is the character $, which matches the last line of the last input file.

2.2.5.3 Context Addresses A context address is a `regular expression' enclosed in slashes (`/'). The regular expressions recognized by sed are the same as those recognized by ed, and are listed in the preceding section. As in the case of ed, for a context address to `match' the input, the whole pattern within the address must match some portion of the pattern space.

2.2.5.4 Number of Addresses The commands which follow can have 0, 1, or 2 addresses. Under each command the maximum number of allowed addresses is given. If a command has no addresses, it is applied to every line in the input. If a command has one address, it is applied to all lines which match that address. If a command has two addresses, it is applied to the first line which matches the first address, and to all subsequent lines until (and including) the first subsequent line which matches the second address. Then an attempt is made on subsequent lines to again match the first address, and the process is repeated. Two addresses are separated by a comma. Here are some examples:

```
/an/        matches lines 1, 3, 4 in our sample text
/an.*an/    matches line 1
/^an/       matches no lines
/./         matches all lines
/\./        matches line 5
/r*an/      matches lines 1,3, 4 (number = zero!)
/\(an\).*\1/matches line 1
```

2.2.6 Functions

All functions are named by a single character. In the following summary, the maximum number of allowable addresses is given enclosed in parentheses, then the single character function name, possible arguments enclosed in angles (< >), an expanded English translation of the single-character name, and finally a description of what each function does. The angles around the arguments are not part of the argument, and should not be typed in actual editing commands.

2.2.6.1 Whole-line Oriented Functions

- (2)d -- delete lines

  The d function deletes from the file (does not write to the output) all those lines matched by its address(es).

It also has the side effect that no further commands are attempted on the corpse of a deleted line; as soon as the *d* function is executed, a new line is read from the input, and the list of editing commands is re-started from the beginning on the new line.

- (2)n -- next line The *n* function reads the next line from the input, replacing the current line. The current line is written to the output if it should be. The list of editing commands is continued following the *n* command.

- (1)a\ <text> -- append lines

  The *a* function causes the argument <text> to be written to the output after the line matched by its address. The *a* command is inherently multi-line; *a* must appear at the end of a line, and <text> may contain any number of lines. To preserve the one-command-to-a-line fiction, the interior newlines must be hidden by a backslash character ('\') immediately preceding the newline. The <text> argument is terminated by the first unhidden newline (the first one not immediately preceded by backslash).

  Once an *a* function is successfully executed, <text> will be written to the output regardless of what later commands do to the line which triggered it. The triggering line may be deleted entirely; <text> will still be written to the output.

  The <text> is not scanned for address matches, and no editing commands are attempted on it. It does not cause any change in the line-number counter.

- (1)i\ <text> -- insert lines

  The *i* function behaves identically to the *a* function, except that <text> is written to the output *before* the matched line. All other comments about the *a* function apply to the *i* function as well.

- (2)c\ <text> -- change lines

  The *c* function deletes the lines selected by its address(es), and replaces them with the lines in <text>. Like *a* and *i*, *c* must be followed by a newline hidden by a backslash; and interior new lines in <text> must be hidden by backslashes.

The _c_ command may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of <text> is written to the output, _not_ one copy per line deleted. As with _a_ and _i_, <text> _is_ not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter.

After a line has been deleted by a _c_ function, no further commands are attempted on it.

If text is appended after a line by _a_ or _r_ functions, and the line is subsequently changed, the text inserted by the _c_ function will be placed _before_ the text of the _a_ or _r_ functions.

Note: Within the text put in the output by these functions, leading blanks and tabs will disappear, as always in sed commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash will not appear in the output.

For example, the list of editing commands:

```
n
a\
XXXX
d
```

applied to our standard input, produces:

```
In Xanadu did Kubhla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n         n
i\        c\
XXXX      XXXX
d
```

2.2.6.2 Substitute Function One very important function changes parts of lines selected by a context search within the line.

(2)s<pattern><replacement><flags> -- substitute

The s function replaces part of a line (selected by <pattern>) with <replacement>. It can best be read:

Substitute for <pattern>, <replacement>

The <pattern> argument contains a pattern, exactly like the patterns in addresses. The only difference between <pattern> and a context address is that the context address must be delimited by slash (`/') characters; <pattern> may be delimited by any character other than space or newline. By default, only the first string matched by <pattern> is replaced, but see the g flag below.

The <replacement> argument begins immediately after the second delimiting character of <pattern>, and must be followed immediately by another instance of the delimiting character. (Thus there are exactly three instances of the delimiting character.) The <replacement> is not a pattern, and the characters which are special in patterns do not have special meaning in <replacement>. Instead, other characters are special:

- & is replaced by the string matched by <pattern>

- \d (where d is a single digit) is replaced by the dth substring matched by parts of <pattern> enclosed in `\(' and `\)'. If nested substrings occur in <pattern>, the dth is determined by counting opening delimiters (`\(').

  As in patterns, special characters may be made literal by preceding them with backslash (`\').

The <flags> argument may contain the following flags:

g -- substitute <replacement> for all (non-overlapping) instances of <pattern> in the line. After a successful substitution, the scan for the next instance of <pattern> begins just after the end of the inserted characters; characters put into the line from <replacement> are not rescanned.

p -- print the line if a successful replacement was done. The p flag causes the line to be written to the output if and only if a substitution was actually

made by the s function. Notice that if several s functions, each followed by a p flag, successfully substitute in the same input line, multiple copies of the line will be written to the output: one for each successful substitution.

w <filename> -- write the line to a file if a successful replacement was done. The w flag causes lines which are actually substituted by the s function to be written to a file named by <filename>. If <filename> exists before Sed is run, it is overwritten; if not, it is created.

A single space must separate w and <filename>.

The possibilities of multiple, somewhat different copies of one input line being written are the same as for p.

A maximum of 10 different file names may be mentioned after w flags and w functions (see below), combined.

Here are some examples. The following command, applied to our standard input,

```
s/to/by/w changes
```

produces, on the standard output:

```
In Xanadu did Kubhla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

and, on the file `changes':

```
Through caverns measureless by man
Down by a sunless sea.
```

If the nocopy option is in effect, the command:

```
s/[.,;?:]/*P&*/gp
```

produces:

```
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
```

Finally, to illustrate the effect of the g flag, the command:

```
/X/s/an/AN/p
```

produces (assuming nocopy mode):

```
In XANadu did Kubhla Khan
```

and the command:

```
/X/s/an/AN/gp
```

produces:

```
In XANadu did Kubhla KhAN
```

2.2.6.3 Input-output Functions

(2)p -- print

The print function writes the addressed lines to the standard output file. They are written at the time the p function is encountered, regardless of what succeeding editing commands may do to the lines.

(2)w <filename> -- write on <filename>

The write function writes the addressed lines to the file named by <filename>. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands may do to them.

Exactly one space must separate the w and <filename>.

A maximum of ten different files may be mentioned in write functions and w flags after s functions, combined.

(1)r <filename> -- read the contents of a file

The read function reads the contents of <filename>, and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing

commands do to the line which matched its address. If r and a functions are executed on the same line, the text from the a functions and the r functions is written to the output in the order that the functions are executed.

Exactly one space must separate the r and <filename>. If a file mentioned by a r function cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

NOTE: Since there is a limit to the number of files that can be opened simultaneously, care should be taken that no more than ten files be mentioned in w functions or flags; that number is reduced by one if any r functions are present. (Only one read file is open at one time.)

Here are some examples. Assume that the file `notel' has the following contents:

```
Note: Kubla Khan (more properly Kublai Khan;
1216-1294) was the grandson and most eminent
successor of Genghiz (Chingiz) Khan, and
founder of the Mongol dynasty in China.
```

Then the following command:

```
/Kubla/r notel
```

produces:

```
In Xanadu did Kubla Khan
    Note: Kubla Khan (more properly Kublai Khan;
    1216-1294) was the grandson and most eminent
    successor of Genghiz (Chingiz) Khan, and
    founder of the Mongol dynasty in China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

2.2.6.4 Multiple Input-line Functions Three functions, all spelled with capital letters, deal specially with pattern spaces containing imbedded newlines; they are intended principally to provide pattern matches across lines in the input.

(2)N -- Next line

The next input line is appended to the current line in the pattern space; the two input lines are separated by an imbedded newline. Pattern matches may extend across the imbedded newline(s).

(2)D -- Delete first part of the pattern space

Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), read another line from the input. In any case, begin the list of editing commands again from its beginning.

(2)P -- Print first part of the pattern space

Print up to and including the first newline in the pattern space.

The _P_ and _D_ functions are equivalent to their lower-case counterparts if there are no imbedded newlines in the pattern space.

2.2.6.5 _Hold and Get Functions_ These functions save and retrieve part of the input for possible later use:

1. (2)h--hold pattern space

   The _h_ functions copies the contents of the pattern space into a hold area (destroying the previous contents of the hold area).

2. (2)H -- Hold pattern space

   The _H_ function appends the contents of the pattern space to the contents of the hold area; the former and new contents are separated by a newline.

3. (2)g -- get contents of hold area

   The _g_ function copies the contents of the hold area into the pattern space (destroying the previous contents of the pattern space).

4. (2)G -- Get contents of hold area

   The _G_ function appends the contents of the hold area to the contents of the pattern space; the former and new contents are separated by a newline.

5. (2)x -- exchange

   The exchange command interchanges the contents of the pattern space and the hold area.

For example, the commands

```
1h
1s/ did.*//
1x
G
s/\n/  :/
```

applied to our standard example, produce:

```
In Xanadu did Kubla Khan  :In Xanadu
A stately pleasure dome decree:  :In Xanadu
Where Alph, the sacred river, ran  :In Xanadu
Through caverns measureless to man  :In Xanadu
Down to a sunless sea.  :In Xanadu
```

2.2.6.6 Flow-of-Control Functions These functions do no editing on the input lines, but control the application of functions to the lines selected by the address part.

(2)! -- Don't

The Don't command causes the next command (written on the same line), to be applied to all and only those input lines not selected by the adress part.

(2){ -- Grouping

The grouping command `{' causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping may appear on the same line as the `{' or on the next line.

The group of commands is terminated by a matching `}' standing on a line by itself.

Groups can be nested.

(0):<label> -- place a label

The label function marks a place in the list of editing commands which may be referred to by b and t functions. The <label> may be any sequence of

eight or fewer characters; if two different colon functions have identical labels, a compile time diagnostic will be generated, and no execution attempted.

(2)b<label> -- branch to label

The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon function with the same <label> was encountered. If no colon function with the same label can be found after all the editing commands have been compiled, a compile time diagnostic is produced, and no execution is attempted. A b function with no <label> is taken to be a branch to the end of the list of editing commands; whatever should be done with the current input line is done, and another input line is read; the list of editing commands is restarted from the beginning on the new line.

(2)t<label> -- test substitutions

The t function tests whether any successful substitutions have been made on the current input line; if so, it branches to <label>; if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset by:

1. reading a new input line, or

2. executing a t function.

2.2.6.7 Miscellaneous Functions

- (1)= -- equals

The = function writes to the standard output the line number of the line matched by its address.

- (1)q -- quit

The q function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.

CHAPTER 3

PATTERN RECOGNITION AND FILE COMPARISON UTILITIES

When preparing documents, it is often necessary to find a string repeated in a file or group of files, in order to make a consistent set of changes, or to compare and contrast two or more files in order to identify the differences between them. In this section, some tools provided by XENIX to accomplish these tasks are compared. Although several of these programs may be used interchangeably, knowing which one will do the job at hand most efficiently is a large part of understanding their use. if the job is planned in advance.

In this chapter more possibilites are introduced for streamlining complicated editing command procedures, and dealing with large numbers of files at once. **Grep,** the first and simplest of these tools, merely prints all lines which match a single specified pattern. A variant of **grep, egrep,** searches for more generalized patterns. **Fgrep** searches for a set of keywords with a particularly fast algorithm. **Grep** and its variations are considered in detail here, along with **awk,** a program which offers some special features, including the capacity to deal with numerics, logical relations, and variables. In addition, **awk** allows for searching particular fields within lines.

Both **grep** and **awk** have as their basis the same principle of pattern recognition as **ed** and **sed.** In each case, a file is searched for the occurrence of a given pattern-- a character or group of characters, a word or word string-- generating a list of contexts where the pattern appears. **Grep,** and the related commands, **egrep** and **fgrep,** are introduced below, followed by a discussion of **awk,** a programming language for carrying out a wide range of complex text manipulation functions.

Also discussed here are three additional programs, **comm,** **diff,** and **diff3,** which compare two or more files and output those lines which are different. In text processing applications these programs can be extremely useful for locating variations between several versions of text. The last tool introduced in this chapter is **spell; spell** allows the user to locate spelling and typographic errors quickly in large quantities of text. <u>Chapter 4</u> contains a detailed summary of the options associated with each of these programs.

3.1 GREP

It is often necessary to find all occurences of some word or pattern in a set of files. The patterns being searched are the same "regular expressions" recognized by the editors, **ed** and **sed**. **Grep** stands for

g/re/p

and does exactly this; it searches and prints every line in a set of files that contains the specified regular expression. Thus,

```
grep 'thing' filel file2 file3 ...
```

finds `thing' wherever it occurs in any of the files `filel', `file2', etc. **Grep** also indicates the file in which the line was found, so that it can be edited later. By using **grep** as a _filter_, a command that reads and transforms input, **grep** can be combined with another shell procedure to become a powerful editing tool. The use of **grep** in shell procedures is discussed at length in _The Programmer's Introduction_.

The commands **grep**, **egrep**, and **fgrep** search a file for a specified pattern. They are expressed in the following form:

```
grep
[ option ] ...
expression [ file ] ...

egrep
[ option ] ...
[ expression ]
[ file ] ...

fgrep
[ option ] ...
[ strings ]
[ file ]
```

Commands of the **grep** family search the input files (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output; unless the -h flag is used, the file name is shown if there is more than one input file.

There are two other members of the **grep** family, **fgrep** and **egrep**. **Grep** patterns are limited regular expressions in the style of **ed**, it uses a compact nondeterministic algorithm. **Egrep** patterns are full regular expressions; it uses a fast

deterministic algorithm that sometimes needs exponential space. Fgrep patterns are fixed strings; grep is fast and compact.

The following options are recognized:

- -v All lines but those matching are printed.
- -c Only a count of matching lines is printed.
- -l The names of files with matching lines are listed (once) separated by newlines.
- -n Each line is preceded by its line number in the file.
- -b Each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context. No output is produced, only status.
- -h Do not print filename headers with output lines.
- -y Alphabetic letters in the pattern will match letters of either case in the input grep and fgrep only).
- -e Same as a simple expression argument, but useful when the expression begins with a -.
- -f The regular expression egrep or string list fgrep is taken from the file.
- -x (Exact) lines matched in their entirety are printed (fgrep only).

Care should be taken when using the characters $ * [^ | ? ' " ( ) and \ in the expression as they are also meaningful to the shell. It is safest to enclose the entire expression argument in single quotes ' '.

Fgrep searches for lines that contain one of the (newline-separated) strings.

Egrep accepts extended regular expressions. In the following description `character' excludes newline:

1. A \ followed by a single character matches that character. The character ^

2. ($) matches the beginning (end) of a line.

3. A . matches any character.

4. A single character not otherwise endowed with special meaning matches that character.

5. A string enclosed in brackets [] matches any single character from the string. Ranges of ASCII character codes may be abbreviated as in `a-z0-9'. A] may occur only as the first character of the string. A literal - must be placed where it can't be mistaken as a range indicator.

6. A regular expression followed by * (+, ?) matches a sequence of 0 or more (1 or more, 0 or 1) matches of the regular expression.

7. Two regular expressions concatenated match a match of the first followed by a match of the second.

8. Two regular expressions separated by | or newline match either a match for the first or a match for the second.

9. A regular expression enclosed in parentheses matches a match for the regular expression.

The order of precedence of operators at the same parenthesis level is [] then ? then concatenation then "|" and newline.

3.2 AWK: A Pattern Scanning and Processing Language

Awk is a programming language designed to make many common information retrieval and text manipulation tasks easy to state and to perform. The basic operation of **awk** is to search input lines consecutively for a match of any patterns which the user has specified. For each pattern, an action can be specified; this action will be performed on each line that matches the pattern.

In **awk** the patterns may be more general than in **grep**, and the actions allowed are more involved than merely printing the matching line. For example, the **awk** program

```
{print $3, $2}
```

prints the third and second columns of a table in that order. The program

```
$2   /A|B|C/
```

prints all input lines with an A, B, or C in the second field. The program

```
$1 != prev       { print; prev = $1 }
```

prints all lines in which the first field is different from the previous first field.

3.2.1 Usage

The command

```
awk  program  [files]
```

executes the **awk** commands in the string program on the set of named files, or on the standard input if there are no files. The statements can also be placed in a file pfile, and executed by the command

```
awk  -f pfile  [files]
```

3.2.2 Program Structure

An **awk** program is a sequence of statements of the form:

```
pattern { action }
pattern { action }
...
```

Each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action is executed. When all the patterns have been tested, the next line is fetched and the matching starts over.

Either the pattern or the action may be left out, but not both. If there is no action for a pattern, the matching line is simply copied to the output. (Thus a line which matches several patterns can be printed several times.) If there is no pattern for an action, then the action is performed for every input line. A line which matches no pattern is ignored.

Since patterns and actions are both optional, actions must be enclosed in braces to distinguish them from patterns.

3.2.3 Records and Fields

Awk input is divided into ``records'' terminated by a record separator. The default record separator is a newline, so by default **awk** processes its input a line at a time. The number of the current record is available in a variable named NR.

Each input record is considered to be divided into ``fields.'' Fields are normally separated by white space-blanks or tabs-but the input field separator may be changed, as described below. Fields are referred to as $1, $2, and so forth, where $1 is the first field, and $0 is the whole input record itself. Fields may be assigned to. The number of fields in the current record is available in a variable named NF.

The variables FS and RS refer to the input field and record separators; they may be changed at any time to any single character. The optional command-line argument -Fc may also be used to set FS to the character **c**.

If the record separator is empty, an empty input line is taken as the record separator, and blanks, tabs and newlines are treated as field separators.

The variable FILENAME contains the name of the current input file.

3.2.4 Printing

An action may have no pattern, in which case the action is executed for all lines. The simplest action is to print some or all of a record; this is accomplished by the **awk** command **print**. This program prints each record, thus copying the input to the output intact. More useful is to print a field or fields from each record. For instance,

```
print $2, $1
```

prints the first two fields in reverse order. Items separated by a comma in the print statement will be separated by the current output field separator when output. Items not separated by commas will be concatenated, so

```
print $1 $2
```

runs the first and second fields together.

The predefined variables NF and NR can be used; for example

```
{ print NR, NF, $0 }
```

prints each record preceded by the record number and the number of fields.

Output may be diverted to multiple files; the program

```
{ print $1 >"foo1"; print $2 >"foo2" }
```

writes the first field, $1, on the file foo1, and the second field on file foo2. The >> notation can also be used:

```
print $1 >>"foo"
```

appends the output to the file foo. (In each case, the output files are created if necessary.) The file name can be a variable or a field as well as a constant; for example,

```
print $1 >$2
```

uses the contents of field 2 as a file name.

Naturally there is a limit on the number of output files; currently it is 10.

Similarly, output can be piped into another process; for instance,

```
print | "mail bwk"
```

mails the output to bwk.

The variables OFS and ORS may be used to change the current output field separator and output record separator. The output record separator is appended to the output of the print statement.

Awk also provides the printf statement for output formatting:

```
printf format expr, expr, ...
```

formats the expressions in the list according to the specification in format and prints them. For example,

```
printf "%8.2f  %10ld\n", $1, $2
```

prints $1 as a floating point number 8 digits wide, with two after the decimal point, and $2 as a 10-digit long decimal number, followed by a newline. No output separators are produced automatically; you must add them yourself, as in this example. The version of printf is identical to that used with C2

3.2.5 Patterns

A pattern in front of an action acts as a selector that determines whether the action is to be executed. A variety of expressions may be used as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary boolean combinations of these.

3.2.6 BEGIN and END

The special pattern BEGIN matches the beginning of the input, before the first record is read. The pattern END matches the end of the input, after the last record has been processed. BEGIN and END thus provide a way to gain control before and after processing, for initialization and wrapup.

As an example, the field separator can be set to a colon by

```
BEGIN   { FS = ":" }
... rest of program ...
```

Or the input lines may be counted by

```
END  { print NR }
```

If BEGIN is present, it must be the first pattern; END must be the last if used.

3.2.7 Regular Expressions

The simplest regular expression is a literal string of characters enclosed in slashes, like

```
/smith/
```

This is actually a complete **awk** program which will print all lines which contain any occurrence of the name ``smith''. If a line contains ``smith'' as part of a larger word, it will also be printed, as in

```
blacksmithing
```

Awk regular expressions include the regular expression forms found in the XENIX text editor **ed** and **grep** (without back-referencing). In addition, **awk** allows parentheses for grouping, | for alternatives, + for ``one or more'', and ? for ``zero or one'', all as in **lex**. Character classes may be abbreviated: [a-zA-Z0-9] is the set of all letters and digits. As an example, the **awk** program

```
/[Aa]ho|[Ww]einberger|[Kk]ernighan/
```

will print all lines which contain any of the names ``Aho,'' ``Weinberger'' or ``Kernighan,'' whether capitalized or not.

Regular expressions (with the extensions listed above) must be enclosed in slashes, just as in **ed** and **sed**. Within a regular expression, blanks and the regular expression metacharacters are significant. To turn of the magic meaning of one of the regular expression characters, precede it with a backslash. An example is the pattern

```
/\/.*\//
```

which matches any string of characters enclosed in slashes.

One can also specify that any field or variable matches a regular expression (or does not match it) with the operators and ! . The program

```
$1   /[jJ]ohn/
```

prints all lines where the first field matches ``john'' or

``John.'' Notice that this will also match ``Johnson'', ``St. Johnsbury'', and so on. To restrict it to exactly [jJ]ohn, use

```
$1   /^[jJ]ohn$/
```

The caret ^ refers to the beginning of a line or field; the dollar sign $ refers to the end.

3.2.8 Relational Expressions

An **awk** pattern can be a relational expression involving the usual relational operators <, <=, ==, !=, >=, and >. An example is

```
$2 > $1 + 100
```

which selects lines where the second field is at least 100 greater than the first field. Similarly,

```
NF % 2 == 0
```

prints lines with an even number of fields.

In relational tests, if neither operand is numeric, a string comparison is made; otherwise it is numeric. Thus,

```
$1 >= "s"
```

selects lines that begin with an s, t, u, etc. In the absence of any other information, fields are treated as strings, so the program

```
$1 > $2
```

will perform a string comparison.

3.2.9 Combinations of Patterns

A pattern can be any boolean combination of patterns, using the operators || (or), && (and), and ! (not). For example,

```
$1 >= "s" && $1 < "t" && $1 != "smith"
```

selects lines where the first field begins with ``s'', but is not ``smith''. && and || guarantee that their operands will be evaluated from left to right; evaluation stops as soon as the truth or falsehood is determined.

3.2.10 Pattern Ranges

The ``pattern'' that selects an action may also consist of two patterns separated by a comma, as in

```
pat1, pat2      { ... }
```

In this case, the action is performed for each line between an occurrence of pat1 and the next occurrence of pat2 (inclusive). For example,

```
/start/, /stop/
```

prints all lines between start and stop, while

```
NR == 100, NR == 200 { ... }
```

does the action for lines 100 through 200 of the input.

3.2.11 Actions

An **awk** action is a sequence of action statements terminated by newlines or semicolons. These action statements can be used to do a variety of bookkeeping and string manipulating tasks.

3.2.12 Built-in Functions

Awk provides a ``length'' function to compute the length of a string of characters. This program prints each record, preceded by its length:

```
{print length, $0}
```

length by itself is a ``pseudo-variable'' which yields the length of the current record; length(argument) is a function which yields the length of its argument, as in the equivalent

```
{print length($0), $0}
```

The argument may be any expression.

Awk also provides the arithmetic functions sqrt, log, exp, and int, for square root, base e logarithm, exponential, and integer part of their respective arguments.

The name of one of these built-in functions, without argument or parentheses, stands for the value of the

function on the whole record. The program

```
length < 10 || length > 20
```

prints lines whose length is less than 10 or greater than 20. The function *substr*(*s*, *m*, *n*) produces the substring of *s* that begins at position *m* (origin 1) and is at most *n* characters long. If *n* is omitted, the substring goes to the end of *s*. The function *index*(*s1*, *s2*) returns the position where the string *s2* occurs in *s1*, or zero if it does not.

The function *sprintf*(*f*, *e1*, *e2*, ...) produces the value of the expressions *e1*, *e2*, etc., in the *printf* format specified by *f*. Thus, for example,

```
x = sprintf("%8.2f %10ld", $1, $2)
```

sets *x* to the string produced by formatting the values of $*1* and $*2*.

3.2.13 Variables, Expressions, and Assignments

Awk variables take on numeric (floating point) or string values according to context. For example, in

```
x = 1
```

x is clearly a number, while in

```
x = "smith"
```

it is clearly a string. Strings are converted to numbers and vice versa whenever context demands it. For instance,

```
x = "3" + "4"
```

assigns 7 to *x*. Strings which cannot be interpreted as numbers in a numerical context will generally have numeric value zero, but it is unwise to count on this behavior.

By default, variables (other than built-ins) are initialized to the null string, which has numerical value zero; this eliminates the need for most *BEGIN* sections. For example, the sums of the first two fields can be computed by

```
        { s1 += $1; s2 += $2 }
END     { print s1, s2 }
```

Arithmetic is done internally in floating point. The arithmetic operators are +, -, *, /, and % (mod). The C

increment ++ and decrement -- operators are also available, and so are the assignment operators +=, -=, *=, /=, and %=. These operators may all be used in expressions.

3.2.14 Field Variables

Fields in **awk** share essentially all of the properties of variables _ they may be used in arithmetic or string operations, and may be assigned to. Thus one can replace the first field with a sequence number like this:

```
{ $1 = NR; print }
```

or accumulate two fields into a third, like this:

```
{ $1 = $2 + $3; print $0 }
```

or assign a string to a field:

```
{ if ($3 > 1000)
        $3 = "too big"
  print
}
```

which replaces the third field by ``too big'' when it is, and in any case prints the record.

Field references may be numerical expressions, as in

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is deemed numeric or string depends on context; in ambiguous cases like

```
if ($1 == $2) ...
```

fields are treated as strings.

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields:

```
n = split(s, array, sep)
```

splits the the string _s_ into _array_[_1_], ..., _array_[_n_]. The number of elements found is returned. If the _sep_ argument is provided, it is used as the field separator; otherwise _FS_ is used as the separator.

3.2.15 String Concatenation

Strings may be concatenated. For example

```
length($1 $2 $3)
```

returns the length of the first three fields. Or in a print statement,

```
print $1 " is " $2
```

prints the two fields separated by `` is ''. Variables and numeric expressions may also appear in concatenations.

3.2.16 Arrays

Array elements are not declared; they spring into existence by being mentioned. Subscripts may have any non-null value, including non-numeric strings. As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input record to the NR-th element of the array x. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the **awk** program

```
        { x[NR] = $0 }
END     { ... program ... }
```

The first action merely records each input line in the array x.

Array elements may be named by non-numeric values, which gives **awk** a capability rather like the associative memory of Snobol tables. Suppose the input contains fields with values like apple, orange, etc. Then the program

```
/apple/ { x["apple"]++ }
/orange/         { x["orange"]++ }
END              { print x["apple"], x["orange"] }
```

increments counts for the named array elements, and prints them at the end of the input.

Any expression can be used as a subscript in an array reference. Thus

```
x[$1] = $2
```

uses the first field of a record (as a string) to index the array x.

Suppose each line of input contains two fields, a name and a non-zero value. Names may be repeated; the task is to print a list of each unique name followed by the sum of all the values for that name. This can be done with the program

```
        { amount[$1] += $2 }
END     { for (name in amount)
                print name, amount[name] }
```

To sort the output, replace the last line by

```
                print name, amount[name] | "sort"
```

3.2.17 Flow-of-Control Statements

Awk provides the basic flow-of-control statements if-else, while, for, and statement grouping with braces, as in C. The if statement was previously introduced without description. The condition in parentheses is evaluated; if it is true, the statement following the if is done. The else part is optional.

The while statement is exactly like that of C. For example, to print all input fields one per line,

```
i = 1
while (i <= NF) {
        print $i
        ++i
}
```

The for statement is also exactly that of C:

```
for (i = 1; i <= NF; i++)
        print $i
```

does the same job as the while statement above.

There is an alternate form of the for statement which is suited for accessing the elements of an associative array:

```
for (i in array)
        statement
```

does statement with i set in turn to each element of array.

The elements are accessed in an apparently random order. Chaos will ensue if *i* is altered, or if any new elements are accessed during the loop.

The expression in the condition part of an *if*, *while* or *for* can include relational operators like <, <=, >, >=, == (``is equal to''), and != (``not equal to''); regular expression matches with the match operators   and ! ; the logical operators ||, &&, and !; and of course parentheses for grouping.

The *break* statement causes an immediate exit from an enclosing *while* or *for*; the *continue* statement causes the next iteration to begin.

The statement *next* causes **awk** to skip immediately to the next record and begin scanning the patterns from the top. The statement *exit* causes the program to behave as if the end of the input had occurred.

Comments may be placed in **awk** programs: they begin with the character # and end with the end of the line, as in

```
print x, y      # this is a comment
```

3.3 DIFF

Diff is a program to compare two files, using the form:

```
diff [-efbh] filel file2
```

Diff tells what lines must be changed in two files to bring them into agreement. If *filel* (file2) is `-', the standard input is used. If *filel* (*file2*) is a directory, then a file in that directory whose file-name is the same as the file-name of *file2* (filel) is used. The normal output contains lines of these forms:

nl a *n3*,*n4*
nl,*n2* d *n3*
nl,*n2* c *n3*,*n4*

These lines resemble **ed** commands to convert *filel* into *file2*. The numbers after the letters pertain to *file2*. In fact, by exchanging `a' for `d' and reading backward one may ascertain equally how to convert *file2* into *filel*. As in *ed*, identical pairs where *nl* = *n2* or *n3* = *n4* are abbreviated as a single number.

Following each of these lines come all the lines that are affected in the first file flagged by `<', then all the lines that are affected in the second file flagged by `>'.

The -b option causes trailing blanks (spaces and tabs) to be ignored and other strings of blanks to compare equal.

The -e option produces a script of *a*, *c* and *d* commands for the editor *ed*, which will recreate *file2* from *file1*. The -f option produces a similar script, not useful with *ed*, in the opposite order. In connection with -e, the following shell program may help maintain multiple versions of a file. Only an ancestral file ($1) and a chain of version-to-version *ed* scripts ($2,$3,...) made by *diff* need be on hand. A `latest version' appears on the standard output. (shift; cat $*; echo '1,$p') | ed - $1

Except in rare circumstances, *diff* finds a smallest sufficient set of file differences.

Option -h does a fast, half-hearted job. It works only when changed stretches are short and well separated, but does work on files of unlimited length. Options -e and -f are unavailable with -h.

3.4 DIFF3

diff3 is a program for 3-way differential file comparison, stated in the form:

```
diff3 [-ex3] file1 file2 file3
```

Diff3 compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

====
all three files differ

====1
file1 is different

====2
file2 is different

====3
file3 is different

The type of change suffered in converting a given range of a given file to some other is indicated in one of these ways:

_f_ : _n1_ a
Text is to be appended after line number
_n1_
in file
_f_,
where
_f_
= 1, 2, or 3.

_f_ : _n1_ , _n2_ c
Text is to be
changed in the range line
_n1_
to line
_n2_.
If
_n1_
=
_n2_,
the range may be abbreviated to
_n1_.

The original contents of the range follows immediately after a **c** indication. When the contents of two files are identical, the contents of the lower-numbered file is suppressed.

Under the -e option, diff3 publishes a script for the editor ed that will incorporate into *file1* all changes between *file2* and *file3*, *i.e.* the changes that normally would be flagged ==== and ====3. Option -x (-3) produces a script to incorporate only changes flagged ==== (====3). The following command will apply the resulting script to `file1'.

```
(cat script; echo '1,$p') | ed - file1
```

3.5 COMM

Comm selects or reject lines common to two sorted files. It is expressed in the form:

```
comm [-[123] ] file1 file2
```

Comm reads *file1* and *file2*, which should be ordered in ASCII collating sequence, and produces a three column output: lines only in *file1*; lines only in *file2*; and lines in both files. The filename `-' means the standard input.

Flags 1, 2, or 3 suppress printing of the corresponding column. Thus comm -12 prints only the lines common to the two files; comm -23 prints only lines in the first file but not in the second; comm -123 is a no-op.

3.6 SPELL

Spell collects words from the specified files, and looks them up in a spelling list. Words that neither occur among nor are derivable (by applying inflections, prefixes or suffixes) from words in the spelling list are printed on the standard output. If no files are named, words are collected from the standard input. **spell** is used with the following format:

```
spell [ option ] ..[file]...

/usr/src/cmd/spell/spellin [list]

/usr/src/cmd/spell/spellout [-d] list
```

Spell ignores most **troff, tbl** and **eqn** constructions. Under the -v option, all words not literally in the spelling list are printed, and plausible derivations from spelling list words are indicated. Under the **-b** option, British spelling is checked. Besides preferring centre, colour, speciality, travelled, etc., this option insists upon -ise in words like standardise, Fowler and the OED to the contrary notwithstanding. Under the **-x** option, every plausible stem is printed with `=' for each word.

The spelling list is based on many sources, and while more haphazard than an ordinary dictionary, is also more effective in respect to proper names and popular technical words. Coverage of the specialized vocabularies of biology, medicine and chemistry is light. Pertinent auxiliary files may be specified by name arguments, indicated below with their default settings. Copies of all output are accumulated in the history file. The stop list filters out misspellings (e.g. thier=thy-y+ier) that would otherwise pass.

Two routines help maintain the hash lists used by **spell.** Both expect a list of words, one per line, from the standard input. **Spellin** adds the words on the standard input to the preexisting list and places a new list on the standard output. If no list is specified, the new list is created from scratch. **Spellout** looks up each word in the standard input and prints on the standard output those that are missing.

CHAPTER 4

TEXT FORMATTING AND DOCUMENT PREPARATION

In addition to the text editors, and pattern recognition and file comparison programs that simplify the work of creating and modifying files for text processing, the XENIX system offers text formatting packages which simplify the production of technical reports, memoranda, formal papers, and documentation, as well as several specialized programs for specifying the final output of tables, mathematical equations, and bibliographic references.

There are two major formatting programs available with XENIX. These programs produce a text with justified right margins, automatic page numbering and titling, automatic hyphenation, and many special features. **nroff** is designed to produce output on terminals and line-printers. **troff** (pronounced ``tee-roff'') instead drives a phototypesetter, which produces very high quality output on photographic paper. This document is itself an example of **troff** output.

4.1 FORMATTING PACKAGES

The basic idea of **nroff** and **troff** is that text is interspersed with ``formatting commands'' that specify in detail how the final output is to look. Typically, these include commands that specify line length, spacing, and running titles.

Because **nroff** and **troff** are relatively hard to learn to use effectively, several ``packages'' of canned formatting requests compatible with **nroff** and **troff** have been designed to allow the user to specify paragraphs, running titles, footnotes, multi-column output, and so on, with less effort and without having to learn **nroff** and **troff**. In this chapter, the ``manuscript'' package known as **-ms** is described in detail. To actually produce a document in standard format using **-ms**, use the command

```
troff -ms files ...
```

for the typesetter, and

```
nroff -ms files ...
```

for a terminal. The **-ms** argument tells troff and nroff to use the manuscript package of formatting requests.

4.2 SUPPORTING TOOLS

In addition to the basic formatters, there are also some supporting programs that aid in document preparation. For example, eqn integrates mathematical symbols and equations into the text of a document. The program **tbl** provides an analogous service for preparing tabular material; it does all the computations necessary to align complicated columns with elements of varying widths. Finally, refer prepares bibliographic citations from a data base, in whatever style is defined by the formatting package. It looks after all the details of numbering references in sequence, filling in page and volume numbers, getting the author's initials and the journal name right, and so on.

4.3 HINTS FOR PREPARING DOCUMENTS

Most documents go through several revisions before they are finally finished; some simple measures will make the work of changing them considerably easier. Since most people change documents by rewriting phrases and adding, deleting or rearranging sentences, subsequent editing of text will be simpler if each sentence starts on a new line, and if each line is short, and breaks at a natural place, such as after a comma or semicolon.

Documents should be broken down into individual files of reasonable size, perhaps ten to fifteen thousand characters. Operations on larger files are considerably slower, and the accidental loss of a small file is less catastrophic than a large one. The files should be split at natural boundaries in the document, and named with conventions that allow them to be processed in groups.

One of the advantages of formatting packages like **-ms** is that they allow formatting decisions to be delayed until the document is printed or typeset. If a document is typed initially with generalized formatting commands like **.PP**, they can be defined appropriately, as necessary, either with a canned package like **-ms**, or with user-defined **nroff** and **troff** commands. If the text has been entered in some systematic way, it is easier to revise.

4.4 A NOTE ABOUT THE PAPERS

What follows is a group of independent papers about **-ms**, the formatting packages **nroff** and **troff**, and some of the specialized formatting programs, including **tbl**, **eqn**, and **refer**. Keep in mind that although these papers were written about UNIX, the operating system from which XENIX is derived, all references to UNIX are equally applicable to XENIX. These papers were written largely by the authors of the programs, using the tools they describe quite extensively. Hence the papers are in themselves excellent examples of the final output of text formatted with these programs.

Typing Documents on the UNIX System: Using the −ms Macros with Troff and Nroff

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This document describes a set of easy-to-use macros for preparing documents on the UNIX system. Documents may be produced on either the phototypesetter or a on a computer terminal, without changing the input.

The macros provide facilities for paragraphs, sections (optionally with automatic numbering), page titles, footnotes, equations, tables, two-column format, and cover pages for papers.

This memo includes, as an appendix, the text of the "Guide to Preparing Documents with −ms" which contains additional examples of features of −ms.

This manual is a revision of, and replaces, "Typing Documents on UNIX," dated November 22, 1974.

November 13, 1978

Typing Documents on the UNIX System: Using the −ms Macros with Troff and Nroff

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction. This memorandum describes a package of commands to produce papers using the *troff* and *nroff* formatting programs on the UNIX system. As with other *roff*-derived programs, text is prepared interspersed with formatting commands. However, this package, which itself is written in *troff* commands, provides higher-level commands than those provided with the basic *troff* program. The commands available in this package are listed in Appendix A.

Text. Type normally, except that instead of indenting for paragraphs, place a line reading ".PP" before each paragraph. This will produce indenting and extra space.

Alternatively, the command .LP that was used here will produce a left-aligned (block) paragraph. The paragraph spacing can be changed: see below under "Registers."

Beginning. For a document with a paper-type cover sheet, the input should start as follows:

```
[optional overall format .RP − see below]
.TL
Title of document (one or more lines)
.AU
Author(s) (may also be several lines)
.AI
Author's institution(s)
.AB
Abstract; to be placed on the cover sheet of a paper.
Line length is 5/6 of normal; use .ll here to change.
.AE (abstract end)
text ... (begins with .PP, which see)
```

To omit some of the standard headings (e.g. no abstract, or no author's institution) just omit the corresponding fields and command lines. The word ABSTRACT can be suppressed by writing ".AB no" for ".AB". Several interspersed .AU and .AI lines can be used for multiple authors. The headings are not compulsory: beginning with a .PP command is perfectly OK and will just start printing an ordinary paragraph. ***Warning:*** You can't just begin a document with a line of text. Some −ms command must precede any text input. When in doubt, use .LP to get proper initialization, although any of the commands .PP, .LP, .TL, .SH, .NH is good enough. Figure 1 shows the legal arrangement of commands at the start of a document.

Cover Sheets and First Pages. The first line of a document signals the general format of the first page. In particular, if it is ".RP" a cover sheet with title and abstract is prepared. The default format is useful for scanning drafts.

In general −ms is arranged so that only one form of a document need be stored, containing all information; the first command gives the format, and unnecessary items for that format are ignored.

Warning: don't put extraneous material between the .TL and .AE commands. Processing of the titling items is special, and other data placed in them may not behave as you expect. Don't forget that some −ms command must precede any input text.

Page headings. The −ms macros, by default, will print a page heading containing a page number (if greater than 1). A default page footer is provided only in *nroff*, where the date is used. The user can make minor adjustments to the page headings/footings by redefining the strings LH, CH, and RH which are the left, center and right portions of the page headings, respectively; and the strings LF, CF, and RF, which are the left, center and right portions of the page footer. For more complex formats, the user can redefine the macros PT and BT, which are invoked respectively at the top and bottom of each page. The margins (taken from registers HM and FM for the top and bottom margin respectively) are normally 1 inch; the page header/footer are in the middle of that space. The user who redefines these macros should be careful not to change parameters such as point size or font without resetting them to default values.

Multi-column formats. If you place the command ".2C" in your document, the document will be printed in double column format beginning at that point. This feature is not too useful in computer terminal output, but is often desirable on the typesetter. The command ".1C" will go back to one-column format and also skip to a new page. The ".2C" command is actually a special case of the command

```
.MC [column width [gutter width]]
```

which makes multiple columns with the specified column and gutter width; as many columns as will fit across the page are used. Thus triple, quadruple, ... column pages can be printed. Whenever the number of columns is changed (except going from full width to some larger number of columns) a new page is started.

Headings. To produce a special heading, there are two commands. If you type

```
.NH
type section heading here
may be several lines
```

you will get automatically numbered section headings (1, 2, 3, ...), in boldface. For example,

```
.NH
Care and Feeding of Department Heads
```

produces

1. Care and Feeding of Department Heads

Alternatively,

```
.SH
Care and Feeding of Directors
```

will print the heading with no number added:

Care and Feeding of Directors

Every section heading, of either type, should be followed by a paragraph beginning with .PP or .LP, indicating the end of the heading. Headings may contain more than one line of text.

The .NH command also supports more complex numbering schemes. If a numerical argument is given, it is taken to be a "level" number and an appropriate sub-section number is generated. Larger level numbers indicate deeper sub-sections, as in this example:

```
.NH
Erie-Lackawanna
.NH 2
Morris and Essex Division
.NH 3
Gladstone Branch
.NH 3
Montclair Branch
.NH 2
Boonton Line
```

generates:

2. Erie-Lackawanna

2.1. Morris and Essex Division

2.1.1. Gladstone Branch

2.1.2. Montclair Branch

2.2. Boonton Line

An explicit ".NH 0" will reset the numbering of level 1 to one, as here:

```
.NH 0
Penn Central
```

1. Penn Central

Indented paragraphs. (Paragraphs with hanging numbers, e.g. references.) The sequence

```
.IP [1]
Text for first paragraph, typed
normally for as long as you would
like on as many lines as needed.
.IP [2]
Text for second paragraph, ...
```

produces

[1] Text for first paragraph, typed normally for as long as you would like on as many lines as needed.

[2] Text for second paragraph, ...

A series of indented paragraphs may be followed by an ordinary paragraph beginning with .PP or .LP, depending on whether you wish indenting or not. The command .LP was used here.

More sophisticated uses of .IP are also possible. If the label is omitted, for example, a plain block indent is produced.

```
.IP
This material will
just be turned into a
block indent suitable for quotations or
such matter.
.LP
```

will produce

> This material will just be turned into a block indent suitable for quotations or such matter.

If a non-standard amount of indenting is required, it may be specified after the label (in character positions) and will remain in effect until the next .PP or .LP. Thus, the general form of the .IP command contains two additional fields: the label and the indenting length. For example,

```
.IP first: 9
Notice the longer label, requiring larger
indenting for these paragraphs.
.IP second:
And so forth.
.LP
```

produces this:

first: Notice the longer label, requiring larger indenting for these paragraphs.

second: And so forth.

It is also possible to produce multiple nested indents; the command .RS indicates that the next .IP starts from the current indentation level. Each .RE will eat up one level of indenting so you should balance .RS and .RE commands. The .RS command should be thought of as "move right" and the .RE command as "move left". As an example

```
.IP 1.
Bell Laboratories
.RS
.IP 1.1
Murray Hill
.IP 1.2
Holmdel
.IP 1.3
Whippany
.RS
.IP 1.3.1
Madison
.RE
.IP 1.4
Chester
.RE
.LP
```

will result in

1. Bell Laboratories
    - 1.1 Murray Hill
    - 1.2 Holmdel
    - 1.3 Whippany
        - 1.3.1 Madison
    - 1.4 Chester

All of these variations on .LP leave the right margin untouched. Sometimes, for purposes such as setting off a quotation, a paragraph indented on both right and left is required.

> A single paragraph like this is obtained by preceding it with .QP. More complicated material (several paragraphs) should be bracketed with .QS and .QE.

Emphasis. To get italics (on the typesetter) or underlining (on the terminal) say

```
.I
as much text as you want
can be typed here
.R
```

as was done for *these three words.* The .R command restores the normal (usually Roman) font. If only one word is to be italicized, it may be just given on the line with the .I command,

```
.I word
```

and in this case no .R is needed to restore the previous font. **Boldface** can be produced by

```
.B
Text to be set in boldface
goes here
.R
```

and also will be underlined on the terminal or line printer. As with .I, a single word can be placed in boldface by placing it on the same line as the .B command.

A few size changes can be specified similarly with the commands .LG (make larger), .SM (make smaller), and .NL (return to normal size). The size change is two points; the commands may be repeated for increased effect (here one .NL canceled two .SM commands).

If actual underlining as opposed to italicizing is required on the typesetter, the command

```
.UL word
```

will underline a word. There is no way to underline multiple words on the typesetter.

Footnotes. Material placed between lines with the commands .FS (footnote) and .FE (footnote end) will be collected, remembered, and finally placed at the bottom of the current page*. By default, footnotes are 11/12th the length of normal text, but this can be changed using the FL register (see below).

Displays and Tables. To prepare displays of lines, such as tables, in which the lines should not be re-arranged, enclose them in the commands .DS and .DE

```
.DS
table lines, like the
examples here, are placed
between .DS and .DE
.DE
```

By default, lines between .DS and .DE are indented and left-adjusted. You can also center lines, or retain the left margin. Lines bracketed by .DS C and .DE commands are centered (and not re-arranged); lines bracketed by .DS L and .DE are left-adjusted, not indented, and not re-arranged. A plain .DS is equivalent to .DS I, which indents and left-adjusts. Thus,

these lines were preceded
by .DS C and followed by
a .DE command;

whereas

these lines were preceded
by .DS L and followed by
a .DE command.

Note that .DS C centers each line; there is a variant .DS B that makes the display into a left-adjusted block of text, and then centers that entire block. Normally a display is kept together, on one page. If you wish to have a long display which may be split across page boundaries, use .CD, .LD, or .ID in place of the commands .DS C, .DS L, or .DS I respectively. An extra argument to the .DS I or .DS command is taken as an amount to indent. Note: it is tempting to assume that .DS R will right adjust lines, but it doesn't work.

Boxing words or lines. To draw rectangular boxes around words the command

```
.BX word
```

will print [word] as shown. The boxes will not be neat on a terminal, and this should not be used as a substitute for italics.

> Longer pieces of text may be boxed by enclosing them with .B1 and .B2:
>
> ```
> .B1
> text...
> .B2
> ```
>
> as has been done here.

Keeping blocks together. If you wish to keep a table or other block of lines together on a page, there are "keep -

* Like this.

release'' commands. If a block of lines preceded by .KS and followed by .KE does not fit on the remainder of the current page, it will begin on a new page. Lines bracketed by .DS and .DE commands are automatically kept together this way. There is also a ''keep floating'' command: if the block to be kept together is preceded by .KF instead of .KS and does not fit on the current page, it will be moved down through the text until the top of the next page. Thus, no large blank space will be introduced in the document.

Nroff/Troff commands. Among the useful commands from the basic formatting programs are the following. They all work with both typesetter and computer terminal output:

.bp - begin new page.
.br - ''break'', stop running text from line to line.
.sp n - insert n blank lines.
.na - don't adjust right margins.

Date. By default, documents produced on computer terminals have the date at the bottom of each page; documents produced on the typesetter don't. To force the date, say ''.DA''. To force no date, say ''.ND''. To lie about the date, say ''.DA July 4, 1776'' which puts the specified date at the bottom of each page. The command

.ND May 8, 1945

in ".RP" format places the specified date on the cover sheet and nowhere else. Place this line before the title.

Signature line. You can obtain a signature line by placing the command .SG in the document. The authors' names will be output in place of the .SG line. An argument to .SG is used as a typing identification line, and placed after the signatures. The .SG command is ignored in released paper format.

Registers. Certain of the registers used by −ms can be altered to change default settings. They should be changed with .nr commands, as with

.nr PS 9

to make the default point size 9 point. If the effect is needed immediately, the normal *troff* command should be used in addition to changing the number register.

Register	Defines	Takes effect	Default
PS	point size	next para.	10
VS	line spacing	next para.	12 pts
LL	line length	next para.	6"
LT	title length	next para.	6"
PD	para. spacing	next para.	0.3 VS
PI	para. indent	next para.	5 ens
FL	footnote length	next FS	11/12 LL
CW	column width	next 2C	7/15 LL
GW	intercolumn gap	next 2C	1/15 LL
PO	page offset	next page	26/27"
HM	top margin	next page	1"
FM	bottom margin	next page	1"

You may also alter the strings LH, CH, and RH which are the left, center, and right headings respectively; and similarly LF, CF, and RF which are strings in the page footer. The page number on *output* is taken from register PN, to permit changing its output style. For more complicated headers and footers the macros PT and BT can be redefined, as explained earlier.

Accents. To simplify typing certain foreign words, strings representing common accent marks are defined. They precede the letter over which the mark is to appear. Here are the strings:

Input	Output	Input	Output
*'e	é	*~a	ã
*`e	è	*Ce	ě
*:u	ü	*,c	ç
*^e	ê		

Use. After your document is prepared and stored on a file, you can print it on a terminal with the command*

nroff −ms file

and you can print it on the typesetter with the command

troff −ms file

(many options are possible). In each case, if your document is stored in several files, just list all the filenames where we have used ''file''. If equations or tables are used, *eqn* and/or *tbl* must be invoked as preprocessors.

* If .2C was used, pipe the *nroff* output through *col;* make the first line of the input ''.pi /usr/bin/col.''

References and further study. If you have to do Greek or mathematics, see ***eqn*** [1] for equation setting. To aid ***eqn*** users, ***−ms*** provides definitions of .EQ and .EN which normally center the equation and set it off slightly. An argument on .EQ is taken to be an equation number and placed in the right margin near the equation. In addition, there are three special arguments to EQ: the letters C, I, and L indicate centered (default), indented, and left adjusted equations, respectively. If there is both a format argument and an equation number, give the format argument first, as in

.EQ L (1.3a)

for a left-adjusted equation numbered (1.3a).

Similarly, the macros .TS and .TE are defined to separate tables (see [2]) from text with a little space. A very long table with a heading may be broken across pages by beginning it with .TS H instead of .TS, and placing the line .TH in the table data after the heading. If the table has no heading repeated from page to page, just use the ordinary .TS and .TE macros.

To learn more about *troff* see [3] for a general introduction, and [4] for the full details (experts only). Information on related UNIX commands is in [5]. For jobs that do not seem well-adapted to −ms, consider other macro packages. It is often far easier to write a specific macro packages for such tasks as imitating particular journals than to try to adapt −ms.

Acknowledgment. Many thanks are due to Brian Kernighan for his help in the design and implementation of this package, and for his assistance in preparing this manual.

References

[1] B. W. Kernighan and L. L. Cherry, *Typesetting Mathematics — Users Guide (2nd edition)*, Bell Laboratories Computing Science Report no. 17.

[2] M. E. Lesk, *Tbl — A Program to Format Tables*, Bell Laboratories Computing Science Report no. 45.

[3] B. W. Kernighan, *A Troff Tutorial*, Bell Laboratories, 1976.

[4] J. F. Ossanna, *Nroff/Troff Reference Manual*, Bell Laboratories Computing Science Report no. 51.

[5] K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual*, Bell Laboratories, 1978.

Appendix A
List of Commands

1C	Return to single column format.
2C	Start double column format.
AB	Begin abstract.
AE	End abstract.
AI	Specify author's institution.
AU	Specify author.
B	Begin boldface.
DA	Provide the date on each page.
DE	End display.
DS	Start display (also CD, LD, ID).
EN	End equation.
EQ	Begin equation.
FE	End footnote.
FS	Begin footnote.
I	Begin italics.
IP	Begin indented paragraph.
KE	Release keep.
KF	Begin floating keep.
KS	Start keep.
LG	Increase type size.
LP	Left aligned block paragraph.
ND	Change or cancel date.
NH	Specify numbered heading.
NL	Return to normal type size.
PP	Begin paragraph.
R	Return to regular font (usually Roman).
RE	End one level of relative indenting.
RP	Use released paper format.
RS	Relative indent increased one level.
SG	Insert signature line.
SH	Specify section heading.
SM	Change to smaller type size.
TL	Specify title.
UL	Underline one word.

Register Names

The following register names are used by −ms internally. Independent use of these names in one's own macros may produce incorrect output. Note that no lower case letters are used in any −ms internal name.

Number registers used in −ms

:	DW	GW	HM	IQ	LL	NA	OJ	PO	T.	TV
#T	EF	H1	HT	IR	LT	NC	PD	PQ	TB	VS
1T	FL	H3	IK	KI	MM	NF	PF	PX	TD	YE
AV	FM	H4	IM	L1	MN	NS	PI	RO	TN	YY
CW	FP	H5	IP	LE	MO	OI	PN	ST	TQ	ZN

String registers used in −ms

'	A5	CB	DW	EZ	I	KF	MR	R1	RT	TL
`	AB	CC	DY	FA	I1	KQ	ND	R2	S0	TM
^	AE	CD	E1	FE	I2	KS	NH	R3	S1	TQ
~	AI	CF	E2	FJ	I3	LB	NL	R4	S2	TS
:	AU	CH	E3	FK	I4	LD	NP	R5	SG	TT
,	B	CM	E4	FN	I5	LG	OD	RC	SH	UL
1C	BG	CS	E5	FO	ID	LP	OK	RE	SM	WB
2C	BT	CT	EE	FQ	IE	ME	PP	RF	SN	WH
A1	C	D	EL	FS	IM	MF	PT	RH	SY	WT
A2	C1	DA	EM	FV	IP	MH	PY	RP	TA	XD
A3	C2	DE	EN	FY	IZ	MN	QF	RQ	TE	XF
A4	CA	DS	EQ	HO	KE	MO	R	RS	TH	XK

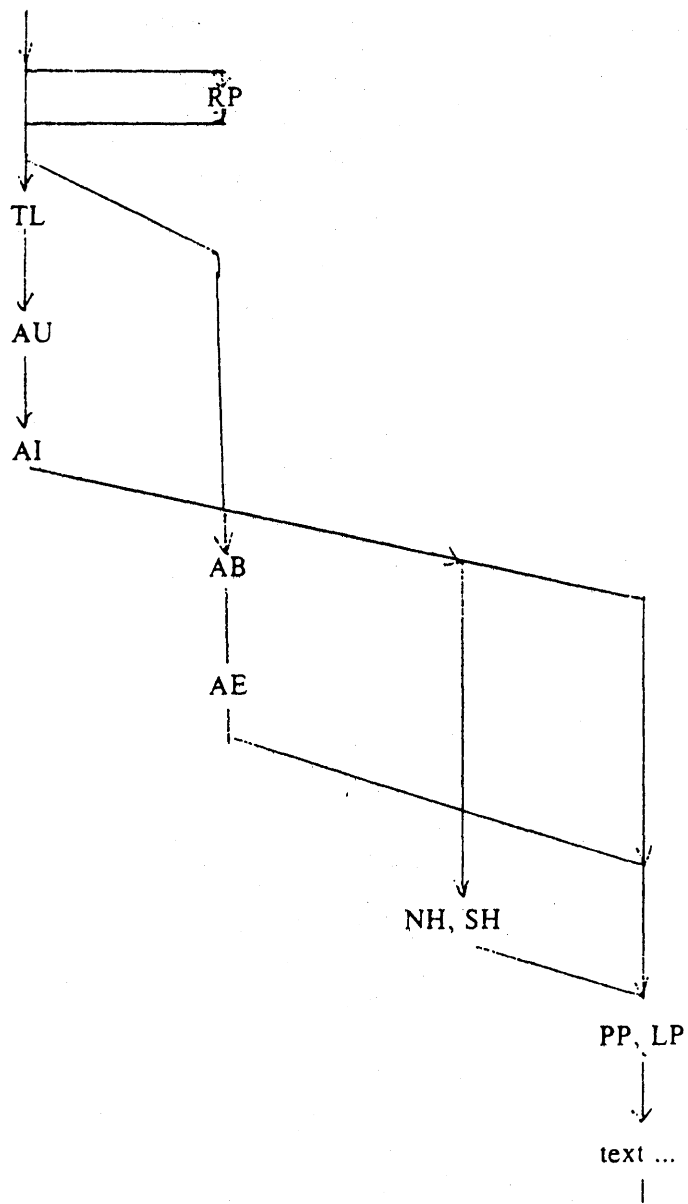


Figure 1

A Guide to Preparing Documents with −ms

M. E. Lesk

Bell Laboratories August 1978

This guide gives some simple examples of document preparation on Bell Labs computers, emphasizing the use of the *−ms* macro package. It enormously abbreviates information in

1. *Typing Documents on UNIX and GCOS*, by M. E. Lesk;
2. *Typesetting Mathematics — User's Guide*, by B. W. Kernighan and L. L. Cherry; and
3. *Tbl — A Program to Format Tables*, by M. E. Lesk.

These memos are all included in the *UNIX Programmer's Manual, Volume 2.* The new user should also have *A Tutorial Introduction to the UNIX Text Editor*, by B. W. Kernighan.

For more detailed information, read *Advanced Editing on UNIX* and *A Troff Tutorial*, by B. W. Kernighan, and (for experts) *Nroff/Troff Reference Manual* by J. F. Ossanna. Information on related commands is found (for UNIX users) in *UNIX for Beginners* by B. W. Kernighan and the *UNIX Programmer's Manual* by K. Thompson and D. M. Ritchie.

Contents

A TM 2
A released paper 3
An internal memo, and headings . . . 4
Lists, displays, and footnotes 5
Indents, keeps, and double column . 6
Equations and registers 7
Tables and usage 8

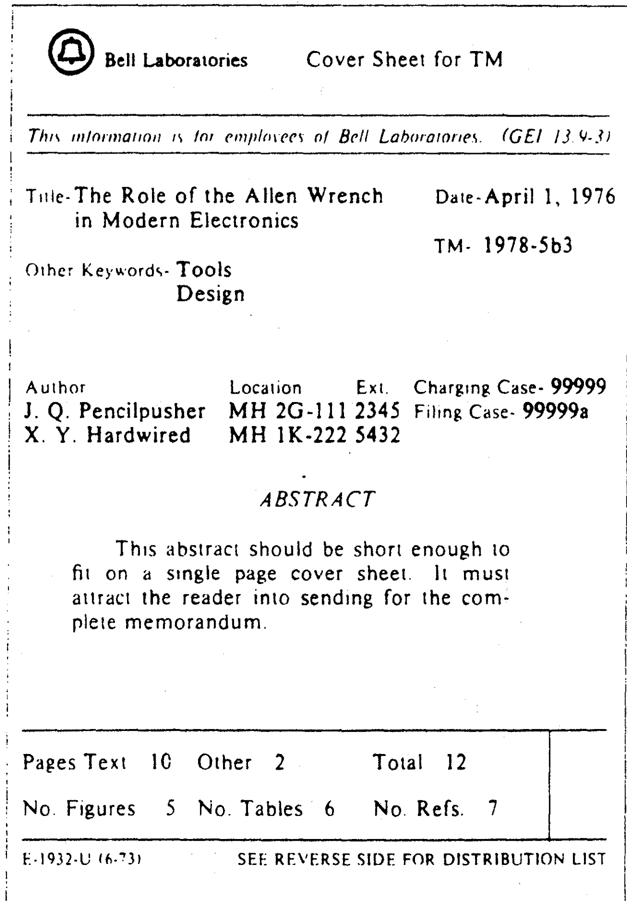
Throughout the examples, input is shown in
  this Helvetica sans serif font
while the resulting output is shown in
  this Times Roman font.

UNIX Document no. 1111

Commands for a TM

```
.TM 1978-5b3 99999 99999-11
.ND April 1, 1976
.TL
The Role of the Allen Wrench in Modern
Electronics
.AU "MH 2G-111" 2345
J. Q. Pencilpusher
.AU "MH 1K-222" 5432
X. Y. Hardwired
.AI
.MH
.OK
Tools
Design
.AB
This abstract should be short enough to
fit on a single page cover sheet.
It must attract the reader into sending for
the complete memorandum.
.AE
.CS 10 2 12 5 6 7
.NH
Introduction.
.PP
Now the first paragraph of actual text ...
...
Last line of text.
.SG MH-1234-JQP/XYH-unix
.NH
References ...
```

Commands not needed in a particular format are ignored.



Bell Laboratories Cover Sheet for TM

This information is for employees of Bell Laboratories. (GEI 13.9-3)

Title- The Role of the Allen Wrench in Modern Electronics Date- April 1, 1976

TM- 1978-5b3

Other Keywords- Tools
Design

Author	Location	Ext.	
J. Q. Pencilpusher	MH 2G-111	2345	Charging Case- 99999
X. Y. Hardwired	MH 1K-222	5432	Filing Case- 99999a

ABSTRACT

This abstract should be short enough to fit on a single page cover sheet. It must attract the reader into sending for the complete memorandum.

Pages Text 10	Other 2	Total 12
No. Figures 5	No. Tables 6	No. Refs. 7

E-1932-U (6-73) SEE REVERSE SIDE FOR DISTRIBUTION LIST

A Released Paper with Mathematics

```
.EQ
delim $$
.EN
.RP

... (as for a TM)

.CS 10 2 12 5 6 7
.NH
Introduction
.PP
The solution to the torque handle equation
.EQ (1)
sum from 0 to inf F ( x sub i ) = G ( x )
.EN
is found with the transformation $ x = rho over
theta $ where $ rho = G prime (x) $ and $theta$
is derived ...
```

The Role of the Allen Wrench
in Modern Electronics

J. Q. Pencilpusher

X. Y. Hardwired

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This abstract should be short enough to fit on a single page cover sheet. It must attract the reader into sending for the complete memorandum.

April 1, 1976

The Role of the Allen Wrench
in Modern Electronics

J. Q. Pencilpusher

X. Y. Hardwired

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

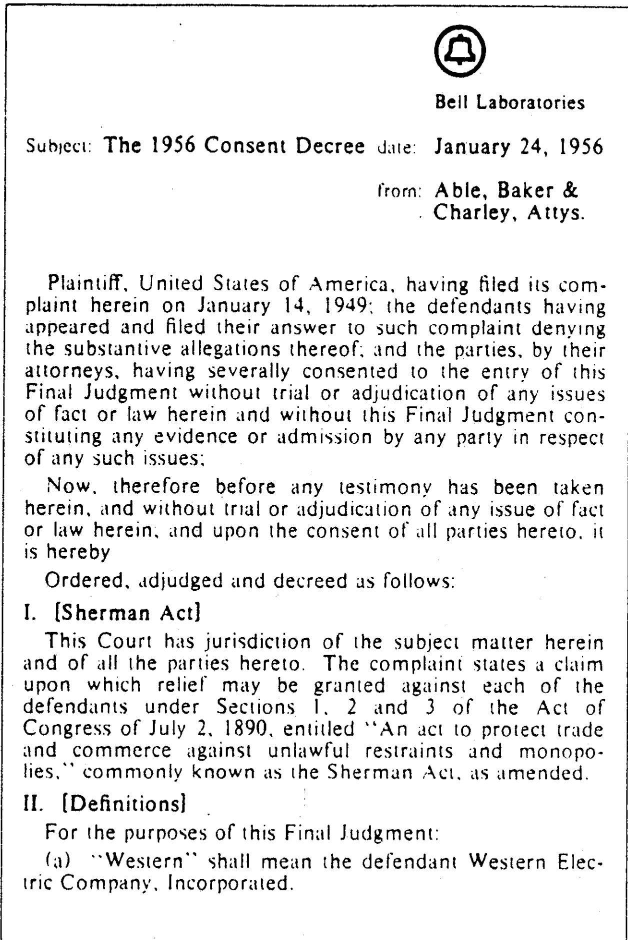
The solution to the torque handle equation

$$\sum_0^\infty F(x_i) = G(x) \qquad (1)$$

is found with the transformation $x = \frac{\rho}{\theta}$ where $\rho = G'(x)$ and θ is derived from well-known principles.

An Internal Memorandum

```
.IM
.ND January 24, 1956
.TL
The 1956 Consent Decree
.AU
Able, Baker &
Charley, Attys.
.PP
Plaintiff, United States of America, having filed
its complaint herein on January 14, 1949; the
defendants having appeared and filed their
answer to such complaint denying the
substantive allegations thereof; and the parties,
by their attorneys, ...
```



Bell Laboratories

Subject: **The 1956 Consent Decree** date: **January 24, 1956**

from: **Able, Baker &**
Charley, Attys.

Plaintiff, United States of America, having filed its complaint herein on January 14, 1949; the defendants having appeared and filed their answer to such complaint denying the substantive allegations thereof; and the parties, by their attorneys, having severally consented to the entry of this Final Judgment without trial or adjudication of any issues of fact or law herein and without this Final Judgment constituting any evidence or admission by any party in respect of any such issues;

Now, therefore before any testimony has been taken herein, and without trial or adjudication of any issue of fact or law herein, and upon the consent of all parties hereto, it is hereby

Ordered, adjudged and decreed as follows:

I. [Sherman Act]

This Court has jurisdiction of the subject matter herein and of all the parties hereto. The complaint states a claim upon which relief may be granted against each of the defendants under Sections 1, 2 and 3 of the Act of Congress of July 2, 1890, entitled "An act to protect trade and commerce against unlawful restraints and monopolies," commonly known as the Sherman Act, as amended.

II. [Definitions]

For the purposes of this Final Judgment:

(a) "Western" shall mean the defendant Western Electric Company, Incorporated.

Other formats possible (specify before .TL) are: .MR ("memo for record"), .MF ("memo for file"), .EG ("engineer's notes") and .TR (Computing Science Tech. Report).

Headings

```
.NH
Introduction.
.PP
text text text
```

1. Introduction

text text text

```
.SH
Appendix I
.PP
text text text
```

Appendix I

text text text

A Simple List

```
.IP 1.
J. Pencilpusher and X. Hardwired,
.I
A New Kind of Set Screw,
.R
Proc. IEEE
.B 75
(1976), 23-255.
.IP 2.
H. Nails and R. Irons,
.I
Fasteners for Printed Circuit Boards,
.R
Proc. ASME
.B 23
(1974), 23-24.
.LP (terminates list)
```

1. J. Pencilpusher and X. Hardwired, *A New Kind of Set Screw*, Proc. IEEE **75** (1976), 23-255.
2. H. Nails and R. Irons, *Fasteners for Printed Circuit Boards*, Proc. ASME **23** (1974), 23-24.

Displays

```
text text text text text text
.DS
and now
for something
completely different
.DE
text text text text text text
```

hoboken harrison newark roseville avenue grove street east orange brick church orange highland avenue mountain station south orange maplewood millburn short hills summit new providence

> and now
> for something
> completely different

murray hill berkeley heights gillette stirling millington lyons basking ridge bernardsville far hills peapack gladstone

Options: .DS L: left-adjust; .DS C: line-by-line center; .DS B: make block, then center.

Footnotes

```
Among the most important occupants
of the workbench are the long-nosed pliers.
Without these basic tools*
.FS
* As first shown by Tiger & Leopard
(1975).
.FE
few assemblies could be completed. They may
lack the popular appeal of the sledgehammer
```

Among the most important occupants of the workbench are the long-nosed pliers. Without these basic tools* few assemblies could be completed. They may lack the popular appeal of the sledgehammer

---

* As first shown by Tiger & Leopard (1975).

Multiple Indents

```
This is ordinary text to point out
the margins of the page.
.IP 1.
First level item
.RS
.IP a)
Second level.
.IP b)
Continued here with another second
level item, but somewhat longer.
.RE
.IP 2.
Return to previous value of the
indenting at this point.
.IP 3.
Another
line.
```

This is ordinary text to point out the margins of the page.

1. First level item
   a) Second level.
   b) Continued here with another second level item, but somewhat longer.
2. Return to previous value of the indenting at this point.
3. Another line.

Keeps

Lines bracketed by the following commands are kept together, and will appear entirely on one page:

.KS	not moved	.KF	may float
.KE	through text	.KE	in text

Double Column

```
.TL
The Declaration of Independence
.2C
.PP
When in the course of human events, it becomes
necessary for one people to dissolve the
political bonds which have connected them with
another, and to assume among the powers of the
earth the separate and equal station to which
the laws of Nature and of Nature's God entitle
them, a decent respect to the opinions of
```

The Declaration of Independence

When in the course of human events, it becomes necessary for one people to dissolve the political bonds which have connected them with another, and to assume among the powers of the earth the separate and equal station to which the laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

We hold these truths to be self-evident, that all men are created equal, that they are endowed by their creator with certain unalienable rights, that among these are life, liberty, and the pursuit of happiness. That to secure these rights, governments are instituted among men,

Equations

A displayed equation is marked with an equation number at the right margin by adding an argument to the EQ line:

```
.EQ (1.3)
x sup 2 over a sup 2 ~=~ sqrt {p z sup 2 +qz+r}
.EN
```

A displayed equation is marked with an equation number at the right margin by adding an argument to the EQ line:

$$\frac{x^2}{a^2} = \sqrt{pz^2+qz+r} \qquad (1.3)$$

```
.EQ I (2.2a)
bold V bar sub nu~=~left [ pile {a above b above
c } right ] + left [ matrix { col { A(11) above .
above . } col { . above . above . } col { . above .
above A(33) }} right ] cdot left [ pile { alpha
above beta above gamma } right ]
.EN
```

$$\bar{\mathbf{V}}_\nu = \begin{bmatrix}a\\b\\c\end{bmatrix} + \begin{bmatrix} A(11) & . & . \\ . & . & . \\ . & . & A(33)\end{bmatrix} \cdot \begin{bmatrix}\alpha\\\beta\\\gamma\end{bmatrix} \qquad (2.2a)$$

```
.EQ L
F hat ( chi ) ~ mark = ~ | del V | sup 2
.EN
.EQ L
lineup =~ {left ( {partial V} over {partial x} right )
} sup 2 + { left ( {partial V} over {partial y} right
) } sup 2 ~~~~~~ lambda -> inf
.EN
```

$$\hat{F}(\chi) = |\nabla V|^2$$

$$= \left(\frac{\partial V}{\partial x}\right)^2 + \left(\frac{\partial V}{\partial y}\right)^2 \qquad \lambda \to \infty$$

```
$ a dot $, $ b dotdot$, $ xi tilde times y vec$:
```

$\dot{a}$, $\ddot{b}$, $\tilde{\xi}\times\vec{y}$. (with delim $$ on, see panel 3).

See also the equations in the second table, panel 8.

Some Registers You Can Change

Line length
  .nr LL 7i

Title length
  .nr LT 7i

Point size
  .nr PS 9

Vertical spacing
  .nr VS 11

Column width
  .nr CW 3i

Intercolumn spacing
  .nr GW .5i

Margins — head and foot
  .nr HM .75i
  .nr FM .75i

Paragraph indent
  .nr PI 2n

Paragraph spacing
  .nr PD 0

Page offset
  .nr PO 0.5i

Page heading
  .ds CH Appendix (center)
  .ds RH 7-25-76 (right)
  .ds LH Private (left)

Page footer
  .ds CF Draft
  .ds LF
  .ds RF   similar

Page numbers
  .nr % 3

Tables

```
.TS              (Ⓣ indicates a tab)
allbox;
c s s
c c c
n n n.
AT&T Common Stock
YearⓉPriceⓉDividend
1971Ⓣ41-54Ⓣ$2.60
2Ⓣ41-54Ⓣ2.70
3Ⓣ46-55Ⓣ2.87
4Ⓣ40-53Ⓣ3.24
5Ⓣ45-52Ⓣ3.40
6Ⓣ51-59Ⓣ.95*
.TE
* (first quarter only)
```

AT&T Common Stock		
Year	Price	Dividend
1971	41-54	$2.60
2	41-54	2.70
3	46-55	2.87
4	40-53	3.24
5	45-52	3.40
6	51-59	.95*

* (first quarter only)

The meanings of the key-letters describing the alignment of each entry are:

c	center	n	numerical
r	right-adjust	a	subcolumn
l	left-adjust	s	spanned

The global table options are center, expand, box, doublebox, allbox, tab (x) and linesize (n).

```
.TS          (with delim $$ on, see panel 3)
doublebox, center;
c c
l l.
NameⓉDefinition
.sp
GammaⓉ$GAMMA (z) = int sub 0 sup inf \
  t sup {z-1} e sup -t dt$
SineⓉ$sin (x) = 1 over 2i ( e sup ix - e sup -ix )$
ErrorⓉ$ roman erf (z) = 2 over sqrt pi \
  int sub 0 sup z e sup {-t sup 2} dt$
BesselⓉ$ J sub 0 (z) = 1 over pi \
  int sub 0 sup pi cos ( z sin theta ) d theta $
ZetaⓉ$ zeta (s) = \
  sum from k=1 to inf k sup -s ~( Re~s > 1)$
.TE
```

Name	Definition
Gamma	$\Gamma(z)=\int_0^\infty t^{z-1}e^{-t}dt$
Sine	$\sin(x)=\frac{1}{2i}(e^{ix}-e^{-ix})$
Error	$\mathrm{erf}(z)=\frac{2}{\sqrt{\pi}}\int_0^z e^{-t^2}dt$
Bessel	$J_0(z)=\frac{1}{\pi}\int_0^\pi \cos(z\sin\theta)d\theta$
Zeta	$\zeta(s)=\sum_{k=1}^{\infty}k^{-s} \quad (\mathrm{Re}\ s>1)$

Usage

Documents with just text:
```
troff -ms files
```
With equations only:
```
eqn files | troff -ms
```
With tables only:
```
tbl files | troff -ms
```
With both tables and equations:
```
tbl files|eqn|troff -ms
```

The above generates STARE output on GCOS: replace −st with −ph for typesetter output.

NROFF/TROFF User's Manual

Joseph F. Ossanna

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

NROFF and TROFF are text processors under the PDP-11 UNIX Time-Sharing System[1] that format text for typewriter-like terminals and for a Graphic Systems phototypesetter, respectively. They accept lines of text interspersed with lines of format control information and format the text into a printable, paginated document having a user-designed style. NROFF and TROFF offer unusual freedom in document styling, including: arbitrary style headers and footers; arbitrary style footnotes; multiple automatic sequence numbering for paragraphs, sections, etc; multiple column output; dynamic font and point-size control; arbitrary horizontal and vertical local motions at any point; and 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. Conditional input is provided that enables the user to embed input expressly destined for either program. NROFF can prepare output directly for a variety of terminal types and is capable of utilizing the full resolution of each terminal.

Usage

The general form of invoking NROFF (or TROFF) at UNIX command level is

nroff *options files* (or **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. The options, which may appear in any order so long as they appear before the files, are:

Option	*Effect*
−o*list*	Print only pages whose page numbers appear in *list*, which consists of comma-separated numbers and number ranges. A number range has the form *N−M* and means pages *N* through *M;* a initial −*N* means from the beginning to page *N;* and a final *N*− means from *N* to the end.
−n*N*	Number first generated page *N*.
−s*N*	Stop every *N* pages. NROFF will halt prior to every *N* pages (default *N*=1) to allow paper loading or changing, and will resume upon receipt of a newline. TROFF will stop the phototypesetter every *N* pages, produce a trailer to allow changing cassettes, and will resume after the phototypesetter START button is pressed.
−m*name*	Prepends the macro file **/usr/lib/tmac.***name* to the input *files*.
−r*aN*	Register *a* (one-character) is set to *N*.
−i	Read standard input after the input files are exhausted.
−q	Invoke the simultaneous input-output mode of the **rd** request.

NROFF Only

−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.

TROFF Only

−t Direct output to the standard output instead of the phototypesetter.

−f Refrain from feeding out paper and stopping phototypesetter at the end of the run.

−w Wait until phototypesetter is available, if currently busy.

−b TROFF will report whether the phototypesetter is busy or available. No text processing is done.

−a Send a printable (ASCII) approximation of the results to the standard output.

−p*N* Print all characters in point size *N* while retaining all prescribed spacings and motions, to reduce phototypesetter elasped time.

−g Prepare output for the Murray Hill Computation Center phototypesetter and direct it to the standard output.

Each option is invoked as a separate argument; for example,

nroff −o*4,8−10* **−T***300S* **−m***abc file1 file2*

requests formatting of pages 4, 8, 9, and 10 of a document contained in the files named *file1* and *file2*, specifies the output terminal as a DASI-300S, and invokes the macro package *abc*.

Various pre- and post-processors are available for use with NROFF and TROFF. These include the equation preprocessors NEQN and EQN[2] (for NROFF and TROFF respectively), and the table-construction preprocessor TBL[3]. A reverse-line postprocessor COL[4] is available for multiple-column NROFF output on terminals without reverse-line ability; COL expects the Model 37 Teletype escape sequences that NROFF produces by default. TK[4] is a 37 Teletype simulator postprocessor for printing NROFF output on a Tektronix 4014. TCAT[4] is phototypesetter-simulator postprocessor for TROFF that produces an approximation of phototypesetter output on a Tektronix 4014. For example, in

tbl *files* | **eqn** | **troff −t** *options* | **tcat**

the first | indicates the piping of TBL's output to EQN's input; the second the piping of EQN's output to TROFF's input; and the third indicates the piping of TROFF's output to TCAT. GCAT[4] can be used to send TROFF (**−g**) output to the Murray Hill Computation Center.

The remainder of this manual consists of: a Summary and Index; a Reference Manual keyed to the index; and a set of Tutorial Examples. Another tutorial is [5].

Joseph F. Ossanna

References

[1] K. Thompson, D. M. Ritchie, *UNIX Programmer's Manual*, Sixth Edition (May 1975).

[2] B. W. Kernighan, L. L. Cherry, *Typesetting Mathematics — User's Guide (Second Edition)*, Bell Laboratories internal memorandum.

[3] M. E. Lesk, *Tbl — A Program to Format Tables*, Bell Laboratories internal memorandum.

[4] Internal on-line documentation, on UNIX.

[5] B. W. Kernighan, *A TROFF Tutorial*, Bell Laboratories internal memorandum.

SUMMARY AND INDEX

Request Form	*Initial Value**	*If No Argument*	*Notes#*	*Explanation*
1. General Explanation				
2. Font and Character Size Control				
.ps ±*N*	10 point	previous	E	Point size; also \s±*N*.†
.ss *N*	12/36 em	ignored	E	Space-character size set to *N*/36 em.†
.cs *FNM*	off	-	P	Constant character space (width) mode (font *F*).†
.bd *F N*	off	-	P	Embolden font *F* by *N*−1 units.†
.bd S *F N*	off	-	P	Embolden Special Font when current font is *F*.†
.ft *F*	Roman	previous	E	Change to font *F* = *x*, *xx*, or 1-4. Also **\f***x*, **\f(***xx*, **\f***N*.
.fp *N F*	R,I,B,S	ignored	-	Font named *F* mounted on physical position 1 ≤ *N* ≤ 4.
3. Page Control				
.pl ±*N*	11 in	11 in	v	Page length.
.bp ±*N*	*N*=1	-	B‡,v	Eject current page; next page number *N*.
.pn ±*N*	*N*=1	ignored	-	Next page number *N*.
.po ±*N*	0; 26/27 in	previous	v	Page offset.
.ne *N*	-	*N*=1 *V*	D,v	Need *N* vertical space (*V* = vertical spacing).
.mk *R*	none	internal	D	Mark current vertical place in register *R*.
.rt ±*N*	none	internal	D,v	Return *(upward only)* to marked vertical place.
4. Text Filling, Adjusting, and Centering				
.br	-	-	B	Break.
.fi	fill	-	B,E	Fill output lines.
.nf	fill	-	B,E	No filling or adjusting of output lines.
.ad *c*	adj, both	adjust	E	Adjust output lines with mode *c*.
.na	adjust	-	E	No output line adjusting.
.ce *N*	off	*N*=1	B,E	Center following *N* input text lines.
5. Vertical Spacing				
.vs *N*	1/6in; 12pts	previous	E,p	Vertical base line spacing (*V*).
.ls *N*	*N*=1	previous	E	Output *N*−1 *V*s after each text output line.
.sp *N*	-	*N*=1 *V*	B,v	Space vertical distance *N in either direction.*
.sv *N*	-	*N*=1 *V*	v	Save vertical distance *N*.
.os	-	-	-	Output saved vertical distance.
.ns	space	-	D	Turn no-space mode on.
.rs	-	-	D	Restore spacing; turn no-space mode off.
6. Line Length and Indenting				
.ll ±*N*	6.5 in	previous	E,m	Line length.
.in ±*N*	*N*=0	previous	B,E,m	Indent.
.ti ±*N*	-	ignored	B,E,m	Temporary indent.
7. Macros, Strings, Diversion, and Position Traps				
.de *xx yy*	-	*.yy*=..	-	Define or redefine macro *xx;* end at call of *yy*.
.am *xx yy*	-	*.yy*=..	-	Append to a macro.
.ds *xx string*	-	ignored	-	Define a string *xx* containing *string*.
.as *xx string*	-	ignored	-	Append *string* to string *xx*.

*Values separated by ";" are for NROFF and TROFF respectively.

#Notes are explained at the end of this Summary and Index

†No effect in NROFF.

‡The use of " ' " as control character (instead of ".") suppresses the break function.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.rm *xx*	-	ignored	-	Remove request, macro, or string.
.rn *xx yy*	-	ignored	-	Rename request, macro, or string *xx* to *yy*.
.di *xx*	-	end	D	Divert output to macro *xx*.
.da *xx*	-	end	D	Divert and append to *xx*.
.wh *N xx*	-	-	**v**	Set location trap; negative is w.r.t. page bottom.
.ch *xx N*	-	-	**v**	Change trap location.
.dt *N xx*	-	off	D,**v**	Set a diversion trap.
.it *N xx*	-	off	E	Set an input-line count trap.
.em *xx*	none	none	-	End macro is *xx*.
8. Number Registers				
.nr *R* ±*N M*		-	**u**	Define and set number register *R*; auto-increment by *M*.
.af *R c*	arabic	-	-	Assign format to register *R* (*c*=**1**, **i**, **I**, **a**, **A**).
.rr *R*	-	-	-	Remove register *R*.
9. Tabs, Leaders, and Fields				
.ta *Nt* ...	0.8; 0.5in	none	E,**m**	Tab settings; *left* type, unless *t*=**R**(right), **C**(centered).
.tc *c*	none	none	E	Tab repetition character.
.lc *c*	.	none	E	Leader repetition character.
.fc *a b*	off	off	-	Set field delimiter *a* and pad character *b*.
10. Input and Output Conventions and Character Translations				
.ec *c*	\	\	-	Set escape character.
.eo	on	-	-	Turn off escape character mechanism.
.lg *N*	-; on	on	-	Ligature mode on if *N*>0.
.ul *N*	off	*N*=1	E	Underline (italicize in TROFF) *N* input lines.
.cu *N*	off	*N*=1	E	Continuous underline in NROFF; like **ul** in TROFF.
.uf *F*	Italic	Italic	-	Underline font set to *F* (to be switched to by **ul**).
.cc *c*	.	.	E	Set control character to *c*.
.c2 *c*	'	'	E	Set nobreak control character to *c*.
.tr *abcd....*	none	-	O	Translate *a* to *b*, etc. on output.
11. Local Horizontal and Vertical Motions, and the Width Function				
12. Overstrike, Bracket, Line-drawing, and Zero-width Functions				
13. Hyphenation.				
.nh	hyphenate	-	E	No hyphenation.
.hy *N*	hyphenate	hyphenate	E	Hyphenate; *N* = mode.
.hc *c*	\%	\%	E	Hyphenation indicator character *c*.
.hw *word1* ...		ignored	-	Exception words.
14. Three Part Titles.				
.tl *'left'center'right'*		-	-	Three part title.
.pc *c*	%	off	-	Page number character.
.lt ±*N*	6.5 in	previous	E,**m**	Length of title.
15. Output Line Numbering.				
.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.
16. Conditional Acceptance of Input				
.if *c anything*		-	-	If condition *c* true, accept *anything* as input, for multi-line use \\{*anything*\\}.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.**if** !*c anything*		-	-	If condition *c* false, accept *anything*.
.**if** *N anything*		-	**u**	If expression $N > 0$, accept *anything*.
.**if** !*N anything*		-	**u**	If expression $N \leqslant 0$, 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*.
.**ie** *c anything*		-	**u**	If portion of if-else; all above forms (like **if**).
.**el** *anything*		-	-	Else portion of if-else.

17. Environment Switching.

.**ev** *N*	*N*=0	previous	-	Environment switched (*push down*).

18. Insertions from the Standard Input

.**rd** *prompt*	-	*prompt*=BEL	-	Read insertion.
.**ex**	-	-	-	Exit from NROFF/TROFF.

19. Input/Output File Switching

.**so** *filename*		-	-	Switch source file *(push down)*.
.**nx** *filename*		end-of-file	-	Next file.
.**pi** *program*		-	-	Pipe output to *program* (NROFF only).

20. Miscellaneous

.**mc** *c N*	-	off	E,m	Set margin character *c* and separation *N*.
.**tm** *string*	-	newline	-	Print *string* on terminal (UNIX standard message output).
.**ig** *yy*	-	*.yy*=..	-	Ignore till call of *yy*.
.**pm** *t*	-	all	-	Print macro names and sizes; if *t* present, print only total of sizes.
.**fl**	-	-	B	Flush output buffer.

21. Output and Error Messages

Notes-

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,p,m,u	Default scale indicator; if not specified, scale indicators are *ignored*.

Alphabetical Request and Section Number Cross Reference

ad 4	cc 10	ds 7	fc 9	ie 16	ll 6	nh 13	pi 19	rn 7	ta 9	vs 5
af 8	ce 4	dt 7	fi 4	if 16	ls 5	nm 15	pl 3	rr 8	tc 9	wh 7
am 7	ch 7	ec 10	fl 20	ig 20	lt 14	nn 15	pm 20	rs 5	ti 6	
as 7	cs 2	el 16	fp 2	in 6	mc 20	nr 8	pn 3	rt 3	tl 14	
bd 2	cu 10	em 7	ft 2	it 7	mk 3	ns 5	po 3	so 19	tm 20	
bp 3	da 7	eo 10	hc 13	lc 9	na 4	nx 19	ps 2	sp 5	tr 10	
br 4	de 7	ev 17	hw 13	lg 10	ne 3	os 5	rd 18	ss 2	uf 10	
c2 10	di 7	ex 18	hy 13	li 10	nf 4	pc 14	rm 7	sv 5	ul 10	

Escape Sequences for Characters, Indicators, and Functions

<table>
<tr><th>Section Reference</th><th>Escape Sequence</th><th>Meaning</th></tr>
<tr><td>10.1</td><td>\\</td><td>\ (to prevent or delay the interpretation of \ )</td></tr>
<tr><td>10.1</td><td>\e</td><td>Printable version of the current escape character.</td></tr>
<tr><td>2.1</td><td>\´</td><td>´ (acute accent); equivalent to \(aa</td></tr>
<tr><td>2.1</td><td>\`</td><td>` (grave accent); equivalent to \(ga</td></tr>
<tr><td>2.1</td><td>\−</td><td>− Minus sign in the current font</td></tr>
<tr><td>7</td><td>\.</td><td>Period (dot) (see de)</td></tr>
<tr><td>11.1</td><td>\(space)</td><td>Unpaddable space-size space character</td></tr>
<tr><td>11.1</td><td>\0</td><td>Digit width space</td></tr>
<tr><td>11.1</td><td>\|</td><td>1/6 em narrow space character (zero width in NROFF)</td></tr>
<tr><td>11.1</td><td>\^</td><td>1/12 em half-narrow space character (zero width in NROFF)</td></tr>
<tr><td>4.1</td><td>\&</td><td>Non-printing, zero width character</td></tr>
<tr><td>10.6</td><td>\!</td><td>Transparent line indicator</td></tr>
<tr><td>10.7</td><td>\"</td><td>Beginning of comment</td></tr>
<tr><td>7.3</td><td>\$N</td><td>Interpolate argument $1 \leqslant N \leqslant 9$</td></tr>
<tr><td>13</td><td>\%</td><td>Default optional hyphenation character</td></tr>
<tr><td>2.1</td><td>\(xx</td><td>Character named xx</td></tr>
<tr><td>7.1</td><td>*x, *(xx</td><td>Interpolate string x or xx</td></tr>
<tr><td>9.1</td><td>\a</td><td>Non-interpreted leader character</td></tr>
<tr><td>12.3</td><td>\b'abc...'</td><td>Bracket building function</td></tr>
<tr><td>4.2</td><td>\c</td><td>Interrupt text processing</td></tr>
<tr><td>11.1</td><td>\d</td><td>Forward (down) 1/2 em vertical motion (1/2 line in NROFF)</td></tr>
<tr><td>2.2</td><td>\fx,\f(xx,\fN</td><td>Change to font named x or xx, or position N</td></tr>
<tr><td>11.1</td><td>\h'N'</td><td>Local horizontal motion; move right N (negative left)</td></tr>
<tr><td>11.3</td><td>\kx</td><td>Mark horizontal input place in register x</td></tr>
<tr><td>12.4</td><td>\l'Nc'</td><td>Horizontal line drawing function (optionally with c)</td></tr>
<tr><td>12.4</td><td>\L'Nc'</td><td>Vertical line drawing function (optionally with c)</td></tr>
<tr><td>8</td><td>\nx,\n(xx</td><td>Interpolate number register x or xx</td></tr>
<tr><td>12.1</td><td>\o'abc...'</td><td>Overstrike characters a, b, c, ...</td></tr>
<tr><td>4.1</td><td>\p</td><td>Break and spread output line</td></tr>
<tr><td>11.1</td><td>\r</td><td>Reverse 1 em vertical motion (reverse line in NROFF)</td></tr>
<tr><td>2.3</td><td>\sN,\s±N</td><td>Point-size change function</td></tr>
<tr><td>9.1</td><td>\t</td><td>Non-interpreted horizontal tab</td></tr>
<tr><td>11.1</td><td>\u</td><td>Reverse (up) 1/2 em vertical motion (1/2 line in NROFF)</td></tr>
<tr><td>11.1</td><td>\v'N'</td><td>Local vertical motion; move down N (negative up)</td></tr>
<tr><td>11.2</td><td>\w'string'</td><td>Interpolate width of string</td></tr>
<tr><td>5.2</td><td>\x'N'</td><td>Extra line-space function (negative before, positive after)</td></tr>
<tr><td>12.2</td><td>\zc</td><td>Print c with zero width (without spacing)</td></tr>
<tr><td>16</td><td>\{</td><td>Begin conditional input</td></tr>
<tr><td>16</td><td>\}</td><td>End conditional input</td></tr>
<tr><td>10.7</td><td>\(newline)</td><td>Concealed (ignored) newline</td></tr>
<tr><td>-</td><td>\X</td><td>X, any character not listed above</td></tr>
</table>

The escape sequences \\, \., \", \$, *, \a, \n, \t, and \(newline) are interpreted in *copy mode* (§7.2).

Predefined General Number Registers

Section Reference	*Register Name*	*Description*
3	%	Current page number.
11.2	**ct**	Character type (set by *width* function).
7.4	**dl**	Width (maximum) of last completed diversion.
7.4	**dn**	Height (vertical size) of last completed diversion.
-	**dw**	Current day of the week (1-7).
-	**dy**	Current day of the month (1-31).
11.3	**hp**	Current horizontal place on *input* line.
15	**ln**	Output line number.
-	**mo**	Current month (1-12).
4.1	**nl**	Vertical position of last printed text base-line.
11.2	**sb**	Depth of string below base line (generated by *width* function).
11.2	**st**	Height of string above base line (generated by *width* function).
-	**yr**	Last two digits of current year.

Predefined Read-Only Number Registers

Section Reference	*Register Name*	*Description*
7.3	**.$**	Number of arguments available at the current macro level.
-	**.A**	Set to 1 in TROFF, if **−a** option used; always 1 in NROFF.
11.1	**.H**	Available horizontal resolution in basic units.
-	**.T**	Set to 1 in NROFF, if **−T** option used; always 0 in TROFF.
11.1	**.V**	Available vertical resolution in basic units.
5.2	**.a**	Post-line extra line-space most recently utilized using **\x**′*N*′.
-	**.c**	Number of *lines* read from current input file.
7.4	**.d**	Current vertical place in current diversion; equal to **nl**, if no diversion.
2.2	**.f**	Current font as physical quadrant (1-4).
4	**.h**	Text base-line high-water mark on current page or diversion.
6	**.i**	Current indent.
6	**.l**	Current line length.
4	**.n**	Length of text portion on previous output line.
3	**.o**	Current page offset.
3	**.p**	Current page length.
2.3	**.s**	Current point size.
7.5	**.t**	Distance to the next trap.
4.1	**.u**	Equal to 1 in fill mode and 0 in nofill mode.
5.1	**.v**	Current vertical line spacing.
11.2	**.w**	Width of previous character.
-	**.x**	Reserved version-dependent register.
-	**.y**	Reserved version-dependent register.
7.4	**.z**	Name of current diversion.

REFERENCE MANUAL

1. General Explanation

1.1. Form of input. Input consists of *text lines*, which are destined to be printed, interspersed with *control lines*, which set parameters or otherwise control subsequent processing. Control lines begin with a *control character*—normally . (period) or ′ (acute accent)—followed by a one or two character name that specifies a basic *request* or the substitution of a user-defined *macro* in place of the control line. The 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/macro name by white space (spaces and/or tabs) for esthetic reasons. Names must be followed by either space or newline. Control lines with unrecognized names are ignored.

Various special functions may be introduced anywhere in the input by means of an *escape* character, normally \. For example, the function **\n***R* causes the interpolation of the contents of the *number register R* in place of the function; here *R* is either a single character name as in **\n***x*, or left-parenthesis-introduced, two-character name as in **\n(***xx*.

1.2. Formatter and device resolution. TROFF internally uses 432 units/inch, corresponding to the Graphic Systems 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).

1.3. Numerical parameter input. Both NROFF and TROFF accept numerical input with the appended scale indicators shown in the following table, where *S* is the current type size in points, *V* is the current vertical line spacing in basic units, and *C* is a *nominal character width* in basic units.

Scale Indicator	Meaning	Number of basic units TROFF	NROFF
i	Inch	432	240
c	Centimeter	432×50/127	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 **ll**, **in**, **ti**, **ta**, **lt**, **po**, **mc**, **\h**, and **\l**; *V*s for the vertically-oriented requests and functions **pl**, **wh**, **ch**, **dt**, **sp**, **sv**, **ne**, **rt**, **\v**, **\x**, and **\L**; **p** for the **vs** request; and **u** for the requests **nr**, **if**, and **ie**. *All* other requests ignore any scale indicators. When a number register containing an already appropriately scaled number is interpolated to provide numerical input, the unit scale indicator **u** may need to be appended to prevent an additional inappropriate default scaling.

The number, *N*, may be specified in decimal-fraction form but the parameter finally stored is rounded to an integer number of basic units.

The *absolute position* indicator | may be prepended to a number *N* to generate the distance to the vertical or horizontal place *N*. For vertically-oriented requests and functions, |*N* becomes the distance in basic units from the current vertical place on the page or in a *diversion* (§7.4) to the 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.

1.4. Numerical expressions. Wherever numerical input is expected an expression involving parentheses, the arithmetic operators +, −, /, *, % (mod), and the logical operators <, >, <=, >=, = (or ==), & (and), : (or) may be used. Except where controlled by parentheses, evaluation of expressions is left-to-right; there is no operator precedence. In the case of certain requests, an initial + or − is stripped and interpreted as an increment or decrement indicator respectively. In the presence of default scaling, the desired scale indicator must be attached to *every* number in an expression for which the desired and default scaling differ. For example, if the number register **x** contains 2 and the current point size is 10, then

.ll (4.25i+\nxP+3)/2u

will set the line length to 1/2 the sum of 4.25 inches + 2 picas + 30 points.

1.5. Notation. Numerical parameters are indicated in this manual in two ways. ±*N* means that the argument may take the forms *N*, +*N*, or −*N* and that the corresponding effect is to set the affected parameter to *N*, to increment it by *N*, or to decrement it by *N* respectively. Plain *N* means that an initial algebraic sign is *not* an increment indicator, but merely the sign of *N*. Generally, unreasonable numerical input is either ignored or truncated to a reasonable value. For example, most requests expect to set parameters to non-negative values; exceptions are **sp**, **wh**, **ch**, **nr**, and **if**. The requests **ps**, **ft**, **po**, **vs**, **ls**, **ll**, **in**, and **lt** restore the *previous* parameter value in the *absence* of an argument.

Single character arguments are indicated by single lower case letters and one/two character arguments are indicated by a pair of lower case letters. Character string arguments are indicated by multi-character mnemonics.

2. Font and Character Size Control

2.1. Character set. The TROFF character set consists of the Graphics Systems Commercial II character set plus a Special Mathematical Font character set—each having 102 characters. These character sets are shown in the attached Table I. 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 given in the attached Table II. The three ASCII exceptions are mapped as follows:

ASCII Input		Printed by TROFF	
Character	Name	Character	Name
'	acute accent	’	close quote
`	grave accent	‘	open quote
−	minus	-	hyphen

The characters ', `, and − may be input by \', \`, and \− respectively or by their names (Table II). The ASCII characters @, #, ", ', `, <, >, \, {, }, ~, ^, and _ exist only on the Special Font and are printed as a 1-em space if that Font is not mounted.

NROFF understands the entire TROFF character set, but can in general print only ASCII characters, additional characters as may be available on the output device, such characters as may be able to 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.

2.2. Fonts. The default mounted fonts are Times Roman (**R**), Times Italic (**I**), Times Bold (**B**), and the Special Mathematical Font (**S**) on physical typesetter positions 1, 2, 3, and 4 respectively. These fonts are used in this document. The *current* font, initially Roman, may be changed (among the mounted fonts) by use of the **ft** request, or by imbedding 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/two-character font name or the numerical font position, 1-4. The current font is available (as numerical position) in the read-only number register **.f**.

NROFF understands font control and normally underlines Italic characters (see §10.5).

2.3. Character size. Character point sizes available on the Graphic Systems typesetter are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36. This is a range of 1/12 inch to 1/2 inch. The **ps** request is used to change or restore the point size. Alternatively the point size may be changed between any two characters by imbedding a \s*N* at the desired point to set the size to *N*, or a \s±*N* ($1 \leqslant N \leqslant 9$) to increment/decrement the size by *N*; **\s0** restores the *previous* size. Requested point size values that are between two valid sizes yield the larger of the two. The current size is available in the **.s** register. NROFF ignores type size control.

Request Form	*Initial Value*	*If No Argument*	*Notes**	*Explanation*
.ps ±*N*	10 point	previous	E	Point size set to ±*N*. Alternatively imbed \s*N* or \s±*N*. Any positive size value may be requested; if invalid, the next larger valid size will result, with a maximum of 36. A paired sequence +*N*, −*N* will work because the previous requested value is also remembered. Ignored in NROFF.
.ss *N*	12/36 em	ignored	E	Space-character size is set to *N*/36 ems. This size is the minimum word spacing in adjusted text. Ignored in NROFF.
.cs *F N M*	off	-	P	Constant character space (width) mode is set on for font *F* (if mounted); the width of every character will be taken to be *N*/36 ems. If *M* is absent, the em is that of the character's point size; if *M* is given, the em is *M*-points. All affected characters are centered in this space, including those with an actual width larger than this space. Special Font characters occurring while the current font is *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.
.bd *F N*	off	-	P	The characters in font *F* will be artificially emboldened 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 column heads above were printed with **.bd I 3**. The mode must be still or again in effect when the characters are physically printed. Ignored in NROFF.

*Notes are explained at the end of the Summary and Index above.

.bd S *F N*	off	-	P	The characters in the Special Font will be emboldened whenever the current font is *F*. This manual was printed with **.bd S B 3**. The mode must be still or again in effect when the characters are physically printed.
.ft *F*	Roman	previous	E	Font changed to *F*. Alternatively, imbed **\f***F*. The font name **P** is reserved to mean the previous font.
.fp *N F*	R,I,B,S	ignored	-	Font position. This is a statement that a font named *F* is mounted on position *N* (1-4). It is a fatal error if *F* is not known. The phototypesetter has four fonts physically mounted. Each font consists of a film strip which can be mounted on a numbered quadrant of a wheel. The default mounting sequence assumed by TROFF is R, I, B, and S on positions 1, 2, 3 and 4.

3. Page control

Top and bottom margins are *not* automatically provided; it is conventional to define two *macros* and to set *traps* for them at vertical positions 0 (top) and $-N$ (*N* from the bottom). See §7 and Tutorial Examples §T2. 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, references to the *current diversion* (§7.4) mean that the mechanism being described works during both ordinary and diverted output (the former considered as the top diversion level).

The useable page width on the Graphic Systems phototypesetter is about 7.54 inches, beginning about 1/27 inch from the left edge of the 8 inch wide, continuous roll paper. The physical limitations on NROFF output are output-device dependent.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.pl $\pm N$	11 in	11 in	v	Page length set to $\pm N$. The internal limitation is about 75 inches in TROFF and about 136 inches in NROFF. The current page length is available in the **.p** register.
.bp $\pm N$	*N*=1	-	B*,v	Begin page. The current page is ejected and a new page is begun. If $\pm N$ is given, the new page number will be $\pm N$. Also see request **ns**.
.pn $\pm N$	*N*=1	ignored	-	Page number. The next page (when it occurs) will have the page number $\pm N$. A **pn** must occur before the initial pseudo-page transition to effect the page number of the first page. The current page number is in the **%** register.
.po $\pm N$	0; 26/27 in†	previous	v	Page offset. The current *left margin* is set to $\pm N$. The TROFF initial value provides about 1 inch of paper margin including the physical typesetter margin of 1/27 inch. In TROFF the maximum (line-length)+(page-offset) is about 7.54 inches. See §6. The current page offset is available in the **.o** register.
.ne *N*	-	*N*=1 *V*	D,v	Need *N* vertical space. If the distance, *D*, to the next trap position (see §7.5) 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

*The use of " ' " as control character (instead of ".") suppresses the break function.

†Values separated by ";" are for NROFF and TROFF respectively.

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.

.mk *R* — none — internal — D — Mark the *current* vertical place in an internal register (both associated with the current diversion level), or in register *R*, if given. See **rt** request.

.rt $\pm N$ — none — internal — D,v — Return *upward only* to a marked vertical place in the current diversion. If $\pm N$ (w.r.t. current place) is given, the place is $\pm N$ from the top of the page or diversion or, if N is absent, to a place marked by a previous **mk**. Note that the **sp** request (§5.3) may be used in all cases instead of **rt** by spacing to the absolute place stored in a explicit register; e. g. using the sequence **.mk** *R* ... **.sp** |\n*R*u.

4. Text Filling, Adjusting, and Centering

4.1. Filling and adjusting. Normally, words are collected from input text lines and assembled into a output text line until some word doesn't fit. An attempt is then made the hyphenate the word in effort to assemble a part of it into the output line. The spaces between the words on the output line are then increased to spread out the line to the current *line length* minus any current *indent*. A *word* is any string of characters delimited by the *space* character or the beginning/end of the input line. Any adjacent pair of words that must be kept together (neither split across output lines nor spread apart in the adjustment process) can be tied together by separating them with the *unpaddable space* character "\ " (backslash-space). The adjusted word spacings are uniform in TROFF and the minimum interword spacing can be controlled with the **ss** request (§2). In NROFF, they are normally nonuniform because of quantization to character-size spaces; however, the command line option **−e** causes uniform spacing with full output device resolution. Filling, adjustment, and hyphenation (§13) can all be prevented or controlled. The *text length* on the last line output is available in the **.n** register, and text base-line position on the page for this line is in the **nl** register. The text base-line high-water mark (lowest place) on the current page is in the **.h** register.

An input text line ending with **.**, **?**, or **!** is taken to be the end of a *sentence*, and an additional space character is automatically provided during filling. Multiple inter-word space characters found in the input are retained, except for trailing spaces; initial spaces also cause a *break*.

When filling is in effect, a **\p** may be imbedded or attached to a word to cause a *break* at the *end* of the word and have the resulting output line *spread out* to fill the current line length.

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 **tr** (§10.5).

4.2. Interrupted text. The copying of a input line in *nofill* (non-fill) mode 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.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.br	-	-	B	Break. The filling of the line currently being collected is stopped and the line is output without adjustment. Text lines beginning with space characters and empty text lines (blank lines) also cause a break.

.fi	fill on	-	B,E	Fill subsequent output lines. The register **.u** is 1 in fill mode and 0 in nofill mode.
.nf	fill on	-	B,E	Nofill. Subsequent output lines are *neither* filled *nor* adjusted. Input text lines are copied directly to output lines *without regard* for the current line length.
.ad *c*	adj,both	adjust	E	Line adjustment is begun. If fill mode is not on, adjustment will be deferred until fill mode is back on. If the type indicator *c* is present, the adjustment type is changed as shown in the following table.

Indicator	Adjust Type
l	adjust left margin only
r	adjust right margin only
c	center
b or **n**	adjust both margins
absent	unchanged

.na	adjust	-	E	Noadjust. Adjustment is turned off; the right margin will be ragged. The adjustment type for **ad** is not changed. Output line filling still occurs if fill mode is on.
.ce *N*	off	*N*=1	B,E	Center the next *N* input text lines within the current (line-length minus indent). If *N*=0, any residual count is cleared. A break occurs after each of the *N* input lines. If the input line is too long, it will be left adjusted.

5. Vertical Spacing

5.1. Base-line spacing. 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 (as in this document). The current *V* is available in the **.v** register. Multiple-*V* line separation (e.g. double spacing) may be requested with **ls**.

5.2. Extra line-space. If a word contains a vertically tall construct requiring the output line containing it to have extra vertical space before and/or after it, the *extra-line-space* function **\x**'*N*' can be imbedded in or attached to that word. In this and other functions having a pair of delimiters around their parameter (here '), the delimiter choice is arbitrary, except that it can't look like the continuation of a number expression for *N*. If *N* is negative, the output line containing the word will be preceded by *N* extra vertical space; if *N* is positive, the output line containing the word will be followed by *N* extra vertical space. If successive requests for extra space apply to the same line, the maximum values are used. The most recently utilized post-line extra line-space is available in the **.a** register.

5.3. Blocks of vertical space. A block of vertical space is ordinarily requested using **sp**, which honors the *no-space* mode and which does not space *past* a trap. A contiguous block of vertical space may be reserved using **sv**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.vs *N*	1/6in;12pts	previous	E,**p**	Set vertical base-line spacing size *V*. Transient *extra* vertical space available with **\x**'*N*' (see above).
.ls *N*	*N*=1	previous	E	*Line* spacing set 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.
.sp *N*	-	*N*=1 *V*	B,v	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).
.sv *N*	-	*N*=1 *V*	v	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 **os**). Subsequent **sv** requests will overwrite any still remembered *N*.
.os	-	-	-	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.
.ns	space	-	D	No-space mode turned on. When on, the no-space mode inhibits **sp** requests and **bp** requests *without* a next page number. The no-space mode is turned off when a line of output occurs, or with **rs**.
.rs	space	-	D	Restore spacing. The no-space mode is turned off.
Blank text line.		-	B	Causes a break and output of a blank line exactly like **sp 1**.

6. Line Length and Indenting

The maximum line length for fill mode may be set with **ll**. The indent may be set with **in**; an indent applicable to *only* the *next* output line may be set with **ti**. The line length includes indent space but *not* page offset space. The line-length minus the indent is the basis for centering with **ce**. The effect of **ll**, **in**, or **ti** is delayed, if a partially collected line exists, until after that line is output. In fill mode the length of text on an output line is less than or equal to the line length minus the indent. The current line length and indent are available in registers **.l** and **.i** respectively. The length of *three-part titles* produced by **tl** (see §14) is *independently* set by **lt**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ll $\pm N$	6.5 in	previous	E,**m**	Line length is set to $\pm N$. In TROFF the maximum (line-length)+(page-offset) is about 7.54 inches.
.in $\pm N$	*N*=0	previous	B,E,**m**	Indent is set to $\pm N$. The indent is prepended to each output line.
.ti $\pm N$	-	ignored	B,E,**m**	Temporary indent. The *next* output text line will be indented a distance $\pm N$ with respect to the current indent. The resulting total indent may not be negative. The current indent is not changed.

7. Macros, Strings, Diversion, and Position Traps

7.1. Macros and strings. A *macro* is a named set of arbitrary *lines* that may be invoked by name or with a *trap*. A *string* is a named string of *characters*, *not* including a newline character, that may be interpolated by name at any point. Request, macro, and string names share the *same* name list. Macro and string names may be one or two characters long and may usurp previously defined request, macro, or string names. Any of these entities may be renamed with **rn** or removed with **rm**. Macros are created by **de** and **di**, and appended to by **am** and **da**; **di** and **da** cause normal output to be stored in a macro. Strings are created by **ds** and appended to by **as**. A macro is invoked in the same way as a request; a

control line beginning *.xx* will interpolate the contents of macro *xx*. The remainder of the line may contain up to nine *arguments*. The strings *x* and *xx* are interpolated at any desired point with *x and *(xx respectively. String references and macro invocations may be nested.

7.2. Copy mode input interpretation. During the definition and extension of strings and macros (not by diversion) the input is read in *copy mode*. The input is copied without interpretation *except* that:

- The contents of number registers indicated by **\n** are interpolated.
- Strings indicated by * are interpolated.
- Arguments indicated by **\$** are interpolated.
- Concealed newlines indicated by \(newline) are eliminated.
- Comments indicated by \" are eliminated.
- **\t** and **\a** are interpreted as ASCII horizontal tab and SOH respectively (§9).
- \\ is interpreted as \.
- \\. is interpreted as ".".

These interpretations can be suppressed by prepending a \. For example, since \\ maps into a \, \\n will copy as \n which will be interpreted as a number register indicator when the macro or string is reread.

7.3. Arguments. When a macro is invoked by name, the remainder of the line is taken to contain up to nine arguments. The argument separator is the space character, and arguments may be surrounded by double-quotes to permit imbedded space characters. Pairs of double-quotes may be imbedded in double-quoted arguments to represent a single double-quote. If the desired arguments won't fit on a line, a concealed newline may be used to continue on the next line.

When a macro is invoked the *input level* is *pushed down* and any arguments available at the previous level become unavailable until the macro is completely read and the previous level is restored. A macro's own arguments can be interpolated at *any* point within the macro with **\$***N*, which interpolates the *N*th argument ($1 \leqslant N \leqslant 9$). If an invoked argument doesn't exist, a null string results. For example, the macro *xx* may be defined by

```
.de xx       \"begin definition
Today is \\$1 the \\$2.
..           \"end definition
```

and called by

.xx Monday 14th

to produce the text

Today is Monday the 14th.

Note that the **\$** was concealed in the definition with a prepended \. The number of currently available arguments is in the **.$** register.

No arguments are available at the top (non-macro) level in this implementation. Because string referencing is implemented as a 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.

7.4. Diversions. Processed output may be diverted into a macro for purposes such as footnote processing (see Tutorial §T5) or determining the horizontal and vertical size of some text for conditional changing of pages or columns. A single diversion trap may be set at a specified vertical position. The number registers **dn** and **dl** respectively contain the vertical and horizontal size of the most recently ended diversion. Processed text that is diverted into a macro retains the vertical size of each of its lines when reread in *nofill* mode regardless of the current *V*. Constant-spaced (**cs**) or emboldened (**bd**) text that is diverted can be reread correctly only if these modes are again or still in effect at reread time. One way

to do this is to imbed in the diversion the appropriate **cs** or **bd** requests with the *transparent* mechanism described in §10.6.

Diversions may be nested and certain parameters and registers are associated with the current diversion level (the top non-diversion level may be thought of as the 0th diversion level). These are the diversion trap and associated macro, no-space mode, the internally-saved marked place (see **mk** and **rt**), the current vertical place (**.d** register), the current high-water text base-line (**.h** register), and the current diversion name (**.z** register).

7.5. Traps. Three types of trap mechanisms are available—page traps, a diversion trap, and an input-line-count trap. Macro-invocation traps may be planted using **wh** at any page position including the top. This trap position may be changed using **ch**. Trap positions at or below the bottom of the page have no effect unless 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 (see Tutorial Examples §T5). 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 **dt**. The **.t** register works in a diversion; if there is no subsequent trap a *large* distance is returned. For a description of input-line-count traps, see **it** below.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.de *xx yy*	-	*.yy*=..	-	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 "..".
.am *xx yy*	-	*.yy*=..	-	Append to macro (append version of **de**).
.ds *xx string*	-	ignored	-	Define a string *xx* containing *string*. Any initial double-quote in *string* is stripped off to permit initial blanks.
.as *xx string*	-	ignored	-	Append *string* to string *xx* (append version of **ds**).
.rm *xx*	-	ignored	-	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 *xx yy*	-	ignored	-	Rename request, macro, or string *xx* to *yy*. If *yy* exists, it is first removed.
.di *xx*	-	end	D	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.

.da *xx*	-	end	D	Divert, appending to *xx* (append version of **di**).
.wh *N xx*	-	-	v	Install a trap to invoke *xx* at page position *N;* a *negative N* will be interpreted with respect to the page *bottom*. Any macro previously planted at *N* is replaced by *xx*. A zero *N* refers to the *top* of a page. In the absence of *xx*, the first found trap at *N*, if any, is removed.
.ch *xx N*	-	-	v	Change the trap position for macro *xx* to be *N*. In the absence of *N*, the trap, if any, is removed.
.dt *N xx*	-	off	D,v	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.
.it *N xx*	-	off	E	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 inline or trap-invoked macros.
.em *xx*	none	none	-	The macro *xx* will be invoked when all input has ended. The effect is the same as if the contents of *xx* had been at the end of the last file processed.

8. Number Registers

A variety of parameters are available to the user as predefined, named *number registers* (see Summary and Index, page 7). In addition, the user may define his own named registers. Register names are one or two characters long and *do not* conflict with request, macro, or string names. Except for certain predefined read-only registers, a number register can be read, written, automatically incremented or decremented, and interpolated into the input in a variety of formats. One common use of user-defined registers is to automatically number sections, paragraphs, lines, etc. A number register may be used any time numerical input is expected or desired and may be used in numerical *expressions* (§1.4).

Number registers are created and modified using **nr**, which specifies the name, numerical value, and the auto-increment size. Registers are also modified, if accessed with an auto-incrementing sequence. If the registers *x* and *xx* both contain *N* and have the auto-increment size *M*, the following access sequences have the effect shown:

Sequence	Effect on Register	Value Interpolated
\n*x*	none	*N*
\n(*xx*	none	*N*
\n+*x*	*x* incremented by *M*	*N*+*M*
\n−*x*	*x* decremented by *M*	*N*−*M*
\n+(*xx*	*xx* incremented by *M*	*N*+*M*
\n−(*xx*	*xx* decremented by *M*	*N*−*M*

When interpolated, a number register is converted to decimal (default), decimal with leading zeros, lower-case Roman, upper-case Roman, lower-case sequential alphabetic, or upper-case sequential alphabetic according to the format specified by **af**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.nr *R* ±*N M*		-	u	The number register *R* is assigned the value ±*N* with respect to the previous value, if any. The increment for auto-incrementing is set to *M*.

.af *R c*	arabic	-	-

Assign format *c* to register *R*. The available formats are:

Format	Numbering Sequence
1	0,1,2,3,4,5,...
001	000,001,002,003,004,005,...
i	0,i,ii,iii,iv,v,...
I	0,I,II,III,IV,V,...
a	0,a,b,c,...,z,aa,ab,...,zz,aaa,...
A	0,A,B,C,...,Z,AA,AB,...,ZZ,AAA,...

An arabic format having *N* digits specifies a field width of *N* digits (example 2 above). The read-only registers and the *width* function (§11.2) are always arabic.

.rr *R*	-	ignored	-

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.

9. Tabs, Leaders, and Fields

9.1. Tabs and leaders. The ASCII horizontal tab character and the ASCII SOH (hereafter known as the *leader* character) can both be used to generate either horizontal motion or a string of repeated characters. The length of the generated entity is governed by internal *tab stops* specifiable with **ta**. The default difference is that tabs generate motion and leaders generate a string of periods; **tc** and **lc** offer the choice of repeated character or motion. There are three types of internal tab stops—*left* adjusting, *right* adjusting, and *centering*. In the following table: *D* is the distance from the current position on the *input* line (where a tab or leader was found) to the next tab stop; *next-string* consists of the input characters following the tab (or leader) up to the next tab (or leader) or end of line; and *W* is the width of *next-string*.

Tab type.	Length of motion or repeated characters	Location of *next-string*
Left	D	Following D
Right	$D-W$	Right adjusted within D
Centered	$D-W/2$	Centered on right end of D

The length of generated motion is allowed to be negative, but that of a repeated character string cannot be. Repeated character strings contain an integer number of characters, and any residual distance is prepended as motion. Tabs or leaders found after the last tab stop are ignored, but may be used as *next-string* terminators.

Tabs and leaders are not interpreted in *copy mode.* **t** and **a** always generate a non-interpreted tab and leader respectively, and are equivalent to actual tabs and leaders in *copy mode.*

9.2. Fields. A *field* is contained between a *pair* of *field delimiter* characters, and consists of sub-strings separated by *padding* indicator characters. The field length is the distance on the *input* line from the position where the field begins to the next tab stop. The difference between the total length of all the sub-strings and the field length is incorporated as horizontal padding space that is divided among the indicated padding places. The incorporated padding is allowed to be negative. For example, if the field delimiter is **#** and the padding indicator is **^**, **#**^*xxx*^*right***#** specifies a right-adjusted string with the string *xxx* centered in the remaining space.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ta *Nt* ...	0.8; 0.5in	none	E,m	Set tab stops and types. *t*=**R**, right adjusting; *t*=C, centering; *t* absent, ieft adjusting. TROFF tab stops are preset every 0.5in.; NROFF every 0.8in. The stop values are separated by spaces, and a value preceded by + is treated as an increment to the previous stop value.
.tc *c*	none	none	E	The tab repetition character becomes *c*, or is removed specifying motion.
.lc *c*	.	none	E	The leader repetition character becomes *c*, or is removed specifying motion.
.fc *a b*	off	off	-	The field delimiter is set to *a*; the padding indicator is set to the *space* character or to *b*, if given. In the absence of arguments the field mechanism is turned off.

10. Input and Output Conventions and Character Translations

10.1. Input character translations. Ways of inputting the graphic character set were discussed in §2.1. The ASCII control characters horizontal tab (§9.1), SOH (§9.1), and backspace (§10.3) are discussed elsewhere. 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 **tr** (§10.5). *All* others are ignored.

The *escape* character \ introduces *escape sequences*—causes the following character to mean another character, or to indicate some function. A complete list of such sequences is given in the Summary and Index on page 6. \ should not be confused with the ASCII control character ESC of the same name. The escape character \ can be input with the sequence \\. The escape character can be changed with **ec**, 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 may be turned off with **eo**, and restored with **ec**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ec *c*	\	\	-	Set escape character to \, or to *c*, if given.
.eo	on	-	-	Turn escape mechanism off.

10.2. Ligatures. 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.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.lg *N*	off; on	on	-	Ligature mode is turned on if *N* is absent or non-zero, and turned 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.

10.3. Backspacing, underlining, overstriking, etc. 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 §12.4. A generalized overstriking function is described in §12.1.

NROFF automatically underlines characters in the *underline* font, specifiable with **uf**, normally that on font position 2 (normally Times Italic, see §2.2). In addition to **ft** and **\f***F*, the underline font may be selected by **ul** and **cu**. Underlining is restricted to an output-device-dependent subset of *reasonable* characters.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ul *N*	off	*N*=1	E	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 **tl** (§14) *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.
.cu *N*	off	*N*=1	E	A variant of **ul** that causes *every* character to be underlined in NROFF. Identical to **ul** in TROFF.
.uf *F*	Italic	Italic	-	Underline font set to *F*. In NROFF, *F* may *not* be on position 1 (initially Times Roman).

10.4. 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.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.cc *c*	.	.	E	The basic control character is set to *c*, or reset to ".".
.c2 *c*	´	´	E	The *nobreak* control character is set to *c*, or reset to "´".

10.5. Output translation. One character can be made a stand-in for another character using **tr**. All text processing (e. g. character comparisons) takes place with the input (stand-in) character which appears to have the width of the final character. The graphic translation occurs at the moment of output (including diversion).

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.tr *abcd....*	none	-	O	Translate *a* into *b*, *c* into *d*, etc. If an odd number of characters is given, the last one will be mapped into the space character. To be consistent, a particular translation must stay in effect from *input* to *output* time.

10.6. 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 imbed control lines in a macro created by a diversion.

10.7. Comments and concealed newlines. An uncomfortably long input line that must stay one line (e. g. a string definition, or nofilled 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. Comments may be imbedded at the *end* of any line by prefacing them with \". The newline at the end of a comment cannot be concealed. A line beginning with \" will appear as a blank line and behave like **.sp 1**; a comment can be on a line by itself by beginning the line with .\".

11. Local Horizontal and Vertical Motions, and the Width Function

11.1. Local Motions. The functions **\v**´*N*´ and **\h**´*N*´ can be used for *local* vertical and horizontal motion respectively. The distance *N* may be negative; the *positive* directions are *rightward* and *downward*. A *local* motion is one contained *within* a line. To avoid unexpected vertical dislocations, it is necessary that the *net* vertical local motion within a word in filled text and otherwise within a line balance to zero. The above and certain other escape sequences providing local motion are summarized in the following table.

<table>
<tr><th>Vertical
Local Motion</th><th colspan="2">Effect in
TROFF NROFF</th><th>Horizontal
Local Motion</th><th colspan="2">Effect in
TROFF NROFF</th></tr>
<tr><td>\v'N'</td><td colspan="2">Move distance N</td><td>\h'N'
\(space)
\0</td><td colspan="2">Move distance N
Unpaddable space-size space
Digit-size space</td></tr>
<tr><td>\u
\d
\r</td><td>½ em up
½ em down
1 em up</td><td>½ line up
½ line down
1 line up</td><td>\|
\^</td><td>1/6 em space
1/12 em space</td><td>ignored
ignored</td></tr>
</table>

As an example, E^2 could be generated by the sequence **E\s−2\v'−0.4m'2\v'0.4m'\s+2**; it should be noted in this example that the 0.4 em vertical motions are at the smaller size.

11.2. Width Function. The *width* function **\w'***string***'** generates the numerical width of *string* (in basic units). Size and font changes may be safely imbedded in *string*, and will not affect the current environment. For example, **.ti −\w'1. 'u** could be used to temporarily indent leftward a distance equal to the size of the string "**1.** ".

The width function also sets three number registers. The registers **st** and **sb** are set respectively to the highest and lowest extent of *string* relative to the baseline; then, for example, the total *height* of the string is **\n(stu−\n(sbu**. In TROFF the number register **ct** is set to a value between 0 and 3: 0 means that all of the characters in *string* were short lower case characters without descenders (like **e**); 1 means that at least one character has a descender (like **y**); 2 means that at least one character is tall (like **H**); and 3 means that both tall characters and characters with descenders are present.

11.3. Mark horizontal place. The escape sequence **\k***x* will cause the *current* horizontal position in the *input line* to be stored in register *x*. As an example, the construction **\k***x word***\h'|\n***x***u+2u'** *word* will embolden *word* by backing up to almost its beginning and overprinting it, resulting in ***word***.

12. Overstrike, Bracket, Line-drawing, and Zero-width Functions

12.1. Overstriking. Automatically centered overstriking of up to nine characters is provided by the *overstrike* function **\o'***string***'**. The characters in *string* overprinted with centers aligned; the total width is that of the widest character. *string* should *not* contain local vertical motion. As examples, **\o'e\''** produces é, and **\o'\(mo\(sl'** produces ∉.

12.2. Zero-width characters. The function **\z***c* will output *c* without spacing over it, and can be used to produce left-aligned overstruck combinations. As examples, **\z\(ci\(pl** will produce ⊕, and **\(br\z\(rn\(ul\(br** will produce the smallest possible constructed box □.

12.3. Large Brackets. The Special Mathematical Font contains a number of bracket construction pieces (⎧ ⎩ ⎫ ⎭ ⎨ ⎬ ⎪ ⎢ ⎥ ⎣ ⎦ ⎡ ⎤) that can be combined into various bracket styles. The function **\b'***string***'** may be used to pile up vertically the characters in *string* (the first character on top and the last at the bottom); the characters are vertically separated by 1 em and the total pile is centered 1/2 em above the current baseline (½ line in NROFF). For example, **\b'\(lc\(lf'E\|\b'\(rc\(rf'\x'−0.5m'\x'0.5m'** produces [E].

12.4. Line drawing. The function **\l'***Nc***'** will draw a string of repeated *c*'s towards the right for a distance *N*. (**\l** is \(lower case L). If *c* looks like a continuation of an expression for *N*, it may insulated from *N* with a **\&**. If *c* is not specified, the _ (baseline rule) is used (underline character in NROFF). If *N* is negative, a backward horizontal motion of size *N* is made *before* drawing the string. Any space resulting from *N*/(size of *c*) having a remainder is put at the beginning (left end) of the string. In the case of characters that are designed to be connected such as baseline-rule _, underrule _, and rooten ‾, the remainder space is covered by over-lapping. If *N* is *less* than the width of *c*, a single *c* is centered on a distance *N*. As an example, a macro to underscore a string can be written

```
.de us
\\$1\l'|0\(ul'
..
```

or one to draw a box around a string

```
.de bx
\(br\|\\$1\|\(br\l'|0\(rn'\l'|0\(ul'
..
```

such that

```
.ul "underlined words"
```

and

```
.bx "words in a box"
```

yield underlined words and words in a box.

The function **\L'***Nc*' will draw a vertical line consisting of the (optional) character *c* stacked vertically apart 1 em (1 line in NROFF), with the first two characters overlapped, if necessary, to form a continuous line. The default character is the *box rule* | (**\(br**); the other suitable character is the *bold vertical* | (**\(bv**). The line is begun without any initial motion relative to the current base line. A positive *N* specifies a line drawn downward and a negative *N* 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.

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 1-em vertical spacings. For example the macro

```
.de eb
.sp -1       \"compensate for next automatic base-line spacing
.nf          \"avoid possibly overflowing word buffer
\h'-.5n'\L'|\\nau-1'\l'\\n(.lu+1n\(ul'\L'-|\\nau+1'\l'|0u-.5n\(ul'   \"draw box
.fi
..
```

will draw a box around some text whose beginning vertical place was saved in number register *a* (e. g. using **.mk a**) as done for this paragraph.

13. Hyphenation.

The automatic hyphenation may be switched off and on. When switched on with **hy**, several variants may be set. A *hyphenation indicator* character may be imbedded in a word to specify desired hyphenation points, or may be prepended to suppress hyphenation. In addition, the user may specify a small exception word list.

Only words that consist of a central alphabetic string surrounded by (usually null) non-alphabetic strings are considered candidates for automatic hyphenation. Words that were input containing hyphens (minus), em-dashes (\(em), or hyphenation indicator characters—such as mother-in-law—are *always* subject to splitting after those characters, whether or not automatic hyphenation is on or off.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.nh	hyphenate	-	E	Automatic hyphenation is turned off.
.hy *N*	on, N=1	on, N=1	E	Automatic hyphenation is turned on for $N \geqslant 1$, or off for $N=0$. If $N=2$, *last* lines (ones that will cause a trap) are not hyphenated. For $N=4$ and 8, the last and first two characters respectively of a word are not split off. These values are additive; i. e. $N=14$ will invoke all three restrictions.
.hc *c*	\%	\%	E	Hyphenation indicator character is set to *c* or to the default \%. The indicator does not appear in the output.
.hw *word1* ...		ignored	-	Specify hyphenation points in words with imbedded minus signs. Versions of a word with terminal *s* are

implied; i. e. *dig−it* implies *dig−its*. This list is examined initially ***and*** after each suffix stripping. The space available is small—about 128 characters.

14. Three Part Titles.

The titling function **tl** provides for automatic placement of three fields at the left, center, and right of a line with a title-length specifiable with **lt**. **tl** may be used anywhere, and is independent of the normal text collecting process. A common use is in header and footer macros.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.tl *'left'center'right'*		-	-	The strings *left*, *center*, and *right* are respectively left-adjusted, centered, and right-adjusted in the current title-length. Any of the strings may be empty, and overlapping is permitted. If the page-number character (initially %) is found within any of the fields it is replaced by the current page number having the format assigned to register %. Any character may be used as the string delimiter.
.pc *c*	%	off	-	The page number character is set to *c*, or removed. The page-number register remains %.
.lt ±*N*	6.5 in	previous	E,m	Length of title set to ±*N*. The line-length and the title-length are *independent*. Indents do not apply to titles; page-offsets do.

15. Output Line Numbering.

Automatic sequence numbering of output lines may be requested with **nm**. When in effect, a three-digit, arabic number plus a digit-space is prepended to output text lines. The text lines are
3 thus offset by four digit-spaces, and otherwise retain their line length; a reduction in line length may be desired to keep the right margin aligned with an earlier margin. Blank lines, other vertical spaces, and lines generated by **tl** are *not* numbered. Numbering can be temporarily suspended with
6 **nn**, or with an **.nm** followed by a later **.nm +0**. In addition, a line number indent *I*, and the number-text separation *S* may be specified in digit-spaces. Further, it can be specified that only those line numbers that are multiples of some number *M* are to be printed (the others will appear
9 as blank number fields).

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.nm ±*N M S I*		off	E	Line number mode. If ±*N* is given, line numbering is turned on, and the next output line numbered is numbered ±*N*. Default values are *M*=1, *S*=1, and *I*=0. Parameters corresponding to missing arguments are unaffected; a non-numeric argument is considered missing. In the absence of all arguments, numbering is turned off; the next line number is preserved for possible further use in number register **ln**.
.nn *N*	-	*N*=1	E	The next *N* text output lines are not numbered.

As an example, the paragraph portions of this section are numbered with *M*=3: **.nm 1 3** was placed at the beginning; **.nm** was placed at the end of the first paragraph; and **.nm +0** was placed
12 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, with
15 *M*=5, with spacing *S* untouched, and with the indent *I* set to 3.

16. Conditional Acceptance of Input

In the following, *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.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.if *c anything*		-	-	If condition *c* true, accept *anything* as input; in multi-line case use \{*anything*\}.
.if !*c anything*		-	-	If condition *c* false, accept *anything*.
.if *N anything*		-	**u**	If expression $N > 0$, accept *anything*.
.if !*N anything*		-	**u**	If expression $N \leqslant 0$, 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*.
.ie *c anything*		-	**u**	If portion of if-else; all above forms (like **if**).
.el *anything*		-	-	Else portion of if-else.

The built-in condition names are:

Condition Name	True If
o	Current page number is odd
e	Current page number is even
t	Formatter is TROFF
n	Formatter is NROFF

If the condition *c* is *true*, or if the number *N* is greater than zero, or if the strings compare identically (including motions and character size and font), *anything* is accepted as input. If a ! precedes the condition, number, or string comparison, the sense of the acceptance is reversed.

Any spaces between the condition and the beginning of *anything* are skipped over. The *anything* can be either a single input line (text, macro, or whatever) or a number of input lines. In the multi-line case, the first line must begin with a left delimiter \{ and the last line must end with a right delimiter \}.

The request **ie** (if-else) is identical to **if** except that the acceptance state is remembered. A subsequent and matching **el** (else) request then uses the reverse sense of that state. **ie** - **el** pairs may be nested.

Some examples are:

```
.if e .tl 'Even Page %'''
```

which outputs a title if the page number is even; and

```
.ie \n%>1 \{\
'sp 0.5i
.tl 'Page %'''
'sp |1.2i \}
.el .sp |2.5i
```

which treats page 1 differently from other pages.

17. Environment Switching.

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.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ev *N*	*N*=0	previous	-	Environment switched to environment $0 \leqslant N \leqslant 2$. Switching is done in push-down fashion so that restoring a previous environment *must* be done with **.ev** rather than specific reference.

18. Insertions from the Standard Input

The input can be temporarily switched to the system *standard input* with **rd**, which will switch back when *two* newlines in a row are found (the *extra* blank line is not used). This mechanism is intended for insertions in form-letter-like documentation. On UNIX, the *standard input* can be the user's keyboard, a *pipe*, or a *file*.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.rd *prompt*	-	*prompt*=BEL	-	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*.
.ex	-	-	-	Exit from NROFF/TROFF. Text processing is terminated exactly as if all input had ended.

If insertions are to be taken from the terminal keyboard *while* output is being printed on the terminal, the command line option **−q** will turn off the echoing of keyboard input and prompt only with BEL. The regular input and insertion input *cannot* simultaneously come from the standard input.

As an example, multiple copies of a form letter may be prepared by entering the insertions for all the copies in one file to be used as the standard input, and causing the file containing the letter to reinvoke itself using **nx** (§19); the process would ultimately be ended by an **ex** in the insertion file.

19. Input/Output File Switching

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.so *filename*		-	-	Switch source file. The top input (file reading) level is switched to *filename*. The effect of an **so** encountered in a macro is not felt until the input level returns to the file level. When the new file ends, input is again taken from the original file. **so**'s may be nested.
.nx *filename*		end-of-file	-	Next file is *filename*. The current file is considered ended, and the input is immediately switched to *filename*.
.pi *program*		-	-	Pipe output to *program* (NROFF only). This request must occur *before* any printing occurs. No arguments are transmitted to *program*.

20. Miscellaneous

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.mc *c N*	-	off	E,**m**	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 will

Request Form	Initial Value	If No Argument	Notes	Explanation
				be appended to the line. If *N* is not given, the previous *N* is used; the initial *N* is 0.2 inches in NROFF and 1 em in TROFF. The margin character used with this paragraph was a 12-point box-rule.
.tm *string*	-	newline	-	After skipping initial blanks, *string* (rest of the line) is read in *copy mode* and written on the user's terminal.
.ig *yy*	-	*.yy*=..	-	Ignore input lines. **ig** behaves exactly like **de** (§7) except that the input is discarded. The input is read in *copy mode*, and any auto-incremented registers will be affected.
.pm *t*	-	all	-	Print macros. The names and sizes of all of the defined macros and strings are printed on the user's terminal; if *t* is given, only the total of the sizes is printed. The sizes is given in *blocks* of 128 characters.
.fl	-	-	B	Flush output buffer. Used in interactive debugging to force output.

21. Output and Error Messages.

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

Various *error* conditions may occur during the operation of NROFF and TROFF. Certain less serious errors having only local impact do not cause processing to terminate. Two examples are *word overflow*, caused by a word that is too large to fit into the word buffer (in fill mode), and *line overflow*, caused by an output line that grew too large to fit in the line buffer; in both cases, a message is printed, the offending excess is discarded, and the affected word or line is marked at the point of truncation with a * in NROFF and a ☞ in TROFF. The 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.

TUTORIAL EXAMPLES

T1. Introduction

Although NROFF and TROFF have by design a syntax reminiscent of earlier text processors* with the intent of easing their use, it is almost always necessary to prepare at least a small set of macro definitions to describe most documents. Such common formatting needs as page margins and footnotes are deliberately not built into NROFF and TROFF. Instead, the macro and string definition, number register, diversion, environment switching, page-position trap, and conditional input mechanisms provide the basis for user-defined implementations.

The examples to be discussed are intended to be useful and somewhat realistic, but won't necessarily cover all relevant contingencies. Explicit numerical parameters are used in the examples to make them easier to read and to illustrate typical values. In many cases, number registers would really be used to reduce the number of places where numerical information is kept, and to concentrate conditional parameter initialization like that which depends on whether TROFF or NROFF is being used.

T2. Page Margins

As discussed in §3, *header* and *footer* macros are usually defined to describe the top and bottom page margin areas respectively. A trap is planted at page position 0 for the header, and at $-N$ (N from the page bottom) for the footer. The simplest such definitions might be

```
.de hd          \"define header
'sp 1i
..              \"end definition
.de fo          \"define footer
'bp
..              \"end definition
.wh 0 hd
.wh −1i fo
```

which provide blank 1 inch top and bottom margins. The header will occur on the *first* page, only if the definition and trap exist prior to the initial pseudo-page transition (§3). In fill mode, the output line that springs the footer trap was typically forced out because some part or whole word didn't fit on it. If anything in the footer and header that follows causes a *break*, that word or part word will be forced out. In this and other examples, requests like **bp** and **sp** that normally cause breaks are invoked using the *no-break* control character ' to avoid this. When the header/footer design contains material requiring independent text processing, the environment may be switched, avoiding most interaction with the running text.

A more realistic example would be

```
.de hd               \"header
.if t .tl '\(rn''\(rn'  \"troff cut mark
.if \\n%>1 \{\
'sp |0.5i−1          \"tl base at 0.5i
.tl ''− % −''        \"centered page number
.ps                  \"restore size
.ft                  \"restore font
.vs \}               \"restore vs
'sp |1.0i            \"space to 1.0i
.ns                  \"turn on no-space mode
..
.de fo               \"footer
.ps 10               \"set footer/header size
.ft R                \"set font
.vs 12p              \"set base-line spacing
.if \\n%=1 \{\
'sp |\\n(.pu−0.5i−1  \"tl base 0.5i up
.tl ''− % −'' \}     \"first page number
'bp
..
.wh 0 hd
.wh −1i fo
```

which sets the size, font, and base-line spacing for the header/footer material, and ultimately restores them. The material in this case is a page number at the bottom of the first page and at the top of the remaining pages. If TROFF is used, a *cut mark* is drawn in the form of *root-en*'s at each margin. The **sp**'s refer to absolute positions to avoid dependence on the base-line spacing. Another reason for this in the footer is that the footer is invoked by printing a line whose vertical spacing swept past the trap position by possibly as

*For example: P. A. Crisman, Ed., *The Compatible Time-Sharing System*, MIT Press, 1965, Section AH9.01 (Description of RUNOFF program on MIT's CTSS system).

much as the base-line spacing. The *no-space* mode is turned on at the end of **hd** to render ineffective accidental occurrences of **sp** at the top of the running text.

The above method of restoring size, font, etc. presupposes that such requests (that set *previous* value) are *not* used in the running text. A better scheme is save and restore both the current *and* previous values as shown for size in the following:

```
.de fo
.nr s1 \\n(.s     \"current size
.ps
.nr s2 \\n(.s     \"previous size
. ---             \"rest of footer
..
.de hd
. ---             \"header stuff
.ps \\n(s2        \"restore previous size
.ps \\n(s1        \"restore current size
..
```

Page numbers may be printed in the bottom margin by a separate macro triggered during the footer's page ejection:

```
.de bn            \"bottom number
.tl ''- % -''     \"centered page number
..
.wh -0.5i-1v bn \"tl base 0.5i up
```

T3. Paragraphs and Headings

The housekeeping associated with starting a new paragraph should be collected in a paragraph macro that, for example, does the desired preparagraph spacing, forces the correct font, size, base-line spacing, and indent, checks that enough space remains for *more than one* line, and requests a temporary indent.

```
.de pg            \"paragraph
.br               \"break
.ft R             \"force font,
.ps 10            \"size,
.vs 12p           \"spacing,
.in 0             \"and indent
.sp 0.4           \"prespace
.ne 1+\\n(.Vu \"want more than 1 line
.ti 0.2i          \"temp indent
..
```

The first break in **pg** will force out any previous partial lines, and must occur before the **vs**. The forcing of font, etc. is partly a defense against prior error and partly to permit things like section heading macros to set parameters only once.

The prespacing parameter is suitable for TROFF; a larger space, at least as big as the output device vertical resolution, would be more suitable in NROFF. The choice of remaining space to test for in the **ne** is the smallest amount greater than one line (the **.V** is the available vertical resolution).

A macro to automatically number section headings might look like:

```
.de sc            \"section
. ---             \"force font, etc.
.sp 0.4           \"prespace
.ne 2.4+\\n(.Vu \"want 2.4+ lines
.fi
\\n+S.
..
.nr S 0 1         \"init S
```

The usage is **.sc**, followed by the section heading text, followed by **.pg**. The **ne** test value includes one line of heading, 0.4 line in the following **pg**, and one line of the paragraph text. A word consisting of the next section number and a period is produced to begin the heading line. The format of the number may be set by **af** (§8).

Another common form is the labeled, indented paragraph, where the label protrudes left into the indent space.

```
.de lp            \"labeled paragraph
.pg
.in 0.5i          \"paragraph indent
.ta 0.2i 0.5i     \"label, paragraph
.ti 0
\t\\$1\t\c        \"flow into paragraph
..
```

The intended usage is "**.lp** *label*"; *label* will begin at 0.2 inch, and cannot exceed a length of 0.3 inch without intruding into the paragraph. The label could be right adjusted against 0.4 inch by setting the tabs instead with **.ta 0.4iR 0.5i**. The last line of **lp** ends with **\c** so that it will become a part of the first line of the text that follows.

T4. Multiple Column Output

The production of multiple column pages requires the footer macro to decide whether it was invoked by other than the last column, so that it will begin a new column rather than produce the bottom margin. The header can initialize a column register that the footer will increment and test. The following is arranged for two columns, but is easily modified for more.

```
.de hd          \"header
. ---
.nr cl 0 1      \"init column count
.mk             \"mark top of text
..
.de fo          \"footer
.ie \\n+(cl<2 \{\
.po +3.4i       \"next column; 3.1+0.3
.rt             \"back to mark
.ns \}          \"no-space mode
.el \{\
.po \\nMu       \"restore left margin
. ---
'bp \}
..
.ll 3.1i        \"column width
.nr M \\n(.o    \"save left margin
```

Typically a portion of the top of the first page contains full width text; the request for the narrower line length, as well as another **.mk** would be made where the two column output was to begin.

T5. Footnote Processing

The footnote mechanism to be described is used by imbedding the footnotes in the input text at the point of reference, demarcated by an initial **.fn** and a terminal **.ef**:

```
.fn
Footnote text and control lines...
.ef
```

In the following, footnotes are processed in a separate environment and diverted for later printing in the space immediately prior to the bottom margin. There is provision for the case where the last collected footnote doesn't completely fit in the available space.

```
.de hd          \"header
. ---
.nr x 0 1       \"init footnote count
.nr y 0-\\nb    \"current footer place
.ch fo -\\nbu   \"reset footer trap
.if \\n(dn .fz  \"leftover footnote
..
.de fo          \"footer
.nr dn 0        \"zero last diversion size
.if \\nx \{\
.ev 1           \"expand footnotes in ev1
.nf             \"retain vertical size
.FN             \"footnotes
.rm FN          \"delete it
.if "\\n(.z"fy" .di \"end overflow diversion
.nr x 0         \"disable fx
.ev \}          \"pop environment
. ---
'bp
..
.de fx          \"process footnote overflow
.if \\nx .di fy \"divert overflow
..
.de fn          \"start footnote
.da FN          \"divert (append) footnote
.ev 1           \"in environment 1
.if \\n+x=1 .fs \"if first, include separator
.fi             \"fill mode
..
.de ef          \"end footnote
.br             \"finish output
.nr z \\n(.v    \"save spacing
.ev             \"pop ev
.di             \"end diversion
.nr y -\\n(dn   \"new footer position,
.if \\nx=1 .nr y -(\\n(.v-\\nz) \
                \"uncertainty correction
.ch fo \\nyu    \"y is negative
.if (\\n(nl+1v)>(\\n(.p+\\ny) \
.ch fo \\n(nlu+1v \"it didn't fit
..
.de fs          \"separator
\l'1i'          \"1 inch rule
.br
..
.de fz          \"get leftover footnote
.fn
.nf             \"retain vertical size
.fy             \"where fx put it
.ef
..
.nr b 1.0i      \"bottom margin size
.wh 0 hd        \"header trap
.wh 12i fo      \"footer trap, temp position
.wh -\\nbu fx   \"fx at footer position
.ch fo -\\nbu   \"conceal fx with fo
```

The header **hd** initializes a footnote count register **x**, and sets both the current footer trap position register **y** and the footer trap itself to a nominal position specified in register **b**. In addition, if the register **dn** indicates a leftover footnote, **fz** is invoked to reprocess it. The footnote start macro **fn** begins a diversion (append) in environment 1, and increments the count **x**; if the count is one, the footnote separator **fs** is interpolated. The separator is kept in a separate macro to permit user redefinition. The footnote end macro **ef** restores the previous environment and ends the diversion after saving the spacing size in register **z**. **y** is then decremented by the size of the

footnote, available in **dn**; then on the first footnote, **y** is further decremented by the difference in vertical base-line spacings of the two environments, to prevent the late triggering the footer trap from causing the last line of the combined footnotes to overflow. The footer trap is then set to the lower (on the page) of **y** or the current page position (**nl**) plus one line, to allow for printing the reference line. If indicated by **x**, the footer **fo** rereads the footnotes from **FN** in nofill mode in environment 1, and deletes **FN**. If the footnotes were too large to fit, the macro **fx** will be trap-invoked to redivert the overflow into **fy**, and the register **dn** will later indicate to the header whether **fy** is empty. Both **fo** and **fx** are planted in the nominal footer trap position in an order that causes **fx** to be concealed unless the **fo** trap is moved. The footer then terminates the overflow diversion, if necessary, and zeros **x** to disable **fx**, because the uncertainty correction together with a not-too-late triggering of the footer can result in the footnote rereading finishing before reaching the **fx** trap.

A good exercise for the student is to combine the multiple-column and footnote mechanisms.

T6. The Last Page

After the last input file has ended, NROFF and TROFF invoke the *end macro* (§7), if any, and when it finishes, eject the remainder of the page. During the eject, any traps encountered are processed normally. At the *end* of this last page, processing terminates *unless* a partial line, word, or partial word remains. If it is desired that another page be started, the end-macro

```
.de en          \"end-macro
\c
'bp
..
.em en
```

will deposit a null partial word, and effect another last page.

Table I

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. The Special Mathematical Font was specially prepared for Bell Laboratories by Graphic Systems, Inc. of Hudson, New Hampshire. The Times Roman, Italic, and Bold are among the many standard fonts available from that company.

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
√ ‾ ≥ ≤ ≡ ∼ ≃ ≠ → ← ↑ ↓ × ÷ ± ∪ ∩ ⊂ ⊃ ⊆ ⊇ ∞ ∂
§ ∇ ¬ ∫ ∝ ∅ ∈ ‡ ☞ ☜ Ⓒ | ○ ⎧ ⎨ ⎩ ⎫ ⎬ ⎭ | ⎣ ⎦ ⎡ ⎤ |

Table II

Input Naming Conventions for ′, `, and −
and for Non-ASCII Special Characters

Non-ASCII characters and *minus* on the standard fonts.

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

Non-ASCII characters and ′, `, _, +, −, =, and * on the special font.

The ASCII characters @, #, ", ′, `, <, >, \, {, }, ~, ^, and _ exist *only* on the special font and are printed as a 1-em space if that font is not mounted. The following characters exist only on the special font except for the upper case Greek letter names followed by † which are mapped into upper case English letters in whatever font is mounted on font position one (default Times Roman). The special math plus, minus, and equals are provided to insulate the appearance of equations from the choice of standard fonts.

Char	Input Name	Character Name
+	\(pl	math plus
−	\(mi	math minus
=	\(eq	math equals
*	\(**	math star
§	\(sc	section
´	\(aa	acute accent
`	\(ga	grave accent
_	\(ul	underrule
/	\(sl	slash (matching backslash)
α	\(*a	alpha
β	\(*b	beta
γ	\(*g	gamma
δ	\(*d	delta
ϵ	\(*e	epsilon
ζ	\(*z	zeta
η	\(*y	eta
θ	\(*h	theta
ι	\(*i	iota
κ	\(*k	kappa
λ	\(*l	lambda
μ	\(*m	mu
ν	\(*n	nu
ξ	\(*c	xi
ο	\(*o	omicron
π	\(*p	pi
ρ	\(*r	rho
σ	\(*s	sigma
ς	\(ts	terminal sigma
τ	\(*t	tau
υ	\(*u	upsilon
ϕ	\(*f	phi
χ	\(*x	chi
ψ	\(*q	psi
ω	\(*w	omega
Α	\(*A	Alpha†
Β	\(*B	Beta†

Char	*Input Name*	*Character Name*
Γ	\(*G	Gamma
Δ	\(*D	Delta
Ε	\(*E	Epsilon†
Ζ	\(*Z	Zeta†
Η	\(*Y	Eta†
Θ	\(*H	Theta
Ι	\(*I	Iota†
Κ	\(*K	Kappa†
Λ	\(*L	Lambda
Μ	\(*M	Mu†
Ν	\(*N	Nu†
Ξ	\(*C	Xi
Ο	\(*O	Omicron†
Π	\(*P	Pi
Ρ	\(*R	Rho†
Σ	\(*S	Sigma
Τ	\(*T	Tau†
Υ	\(*U	Upsilon
Φ	\(*F	Phi
Χ	\(*X	Chi†
Ψ	\(*Q	Psi
Ω	\(*W	Omega
√	\(sr	square root
‾	\(rn	root en extender
⩾	\(>=	>=
⩽	\(<=	<=
≡	\(==	identically equal
≃	\(~=	approx =
~	\(ap	approximates
≠	\(!=	not equal
→	\(->	right arrow
←	\(<-	left arrow
↑	\(ua	up arrow
↓	\(da	down arrow
×	\(mu	multiply
÷	\(di	divide
±	\(+-	plus-minus
∪	\(cu	cup (union)
∩	\(ca	cap (intersection)
⊂	\(sb	subset of
⊃	\(sp	superset of
⊆	\(ib	improper subset
⊇	\(ip	improper superset
∞	\(if	infinity
∂	\(pd	partial derivative
∇	\(gr	gradient
¬	\(no	not
∫	\(is	integral sign
∝	\(pt	proportional to
∅	\(es	empty set
∈	\(mo	member of

Char	*Input Name*	*Character Name*
|	\(br	box vertical rule
‡	\(dd	double dagger
☞	\(rh	right hand
☜	\(lh	left hand
	\(bs	Bell System logo
¦	\(or	or
○	\(ci	circle
⎧	\(lt	left top of big curly bracket
⎩	\(lb	left bottom
⎫	\(rt	right top
⎭	\(rb	right bot
⎨	\(lk	left center of big curly bracket
⎬	\(rk	right center of big curly bracket
⎪	\(bv	bold vertical
⌊	\(lf	left floor (left bottom of big square bracket)
⌋	\(rf	right floor (right bottom)
⌈	\(lc	left ceiling (left top)
⌉	\(rc	right ceiling (right top)

Summary of Changes to N/TROFF Since October 1976 Manual

Options

-h	(Nroff only) Output tabs used during horizontal spacing to speed output as well as reduce output byte count. Device tab settings assumed to be every 8 nominal character widths. The default settings of input (logical) tabs is also initialized to every 8 nominal character widths.
-z	Efficiently suppresses formatted output. Only message output will occur (from "tm"s and diagnostics).

Old Requests

.ad c	The adjustment type indicator "c" may now also be a number previously obtained from the ".j" register (see below).
.so name	The contents of file "name" will be interpolated at the point the "so" is encountered. Previously, the interpolation was done upon return to the file-reading input level.

New Request

.ab text — Prints "text" on the message output and terminates without further processing. If "text" is missing, "User Abort." is printed. Does not cause a break. The output buffer is flushed.

.fz F N — forces font "F" to be in size N. N may have the form N, +N, or -N. For example,

    .fz 3 -2

will cause an implicit \s-2 every time font 3 is entered, and a corresponding \s+2 when it is left. Special font characters occurring during the reign of font F will have the same size modification. If special characters are to be treated differently,

    .fz S F N

may be used to specify the size treatment of special characters during font F. For example,

    .fz 3 -3
    .fz S 3 -0

will cause automatic reduction of font 3 by 3 points while the special characters would not be affected. Any ".fp" request specifying a font on some position must precede ".fz" requests relating to that position.

New Predefined Number Registers.

.k	Read-only. Contains the horizontal size of the text portion (without indent) of the current partially collected output line, if any, in the current environment.
.j	Read-only. A number representing the current adjustment mode and type. Can be saved and later given to the "ad" request to restore a previous mode.
.P	Read-only. 1 if the current page is being printed, and zero otherwise.
.L	Read-only. Contains the current line-spacing parameter ("ls").
c.	General register access to the input line-number in the current input file. Contains the same value as the read-only ".c" register.

A TROFF Tutorial

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

troff is a text-formatting program for driving the Graphic Systems phototypesetter on the UNIX† and GCOS operating systems. This device is capable of producing high quality text; this paper is an example of **troff** output.

The phototypesetter itself normally runs with four fonts, containing roman, italic and bold letters (as on this page), a full greek alphabet, and a substantial number of special characters and mathematical symbols. Characters can be printed in a range of sizes, and placed anywhere on the page.

troff allows the user full control over fonts, sizes, and character positions, as well as the usual features of a formatter — right-margin justification, automatic hyphenation, page titling and numbering, and so on. It also provides macros, arithmetic variables and operations, and conditional testing, for complicated formatting tasks.

This document is an introduction to the most basic use of **troff**. It presents just enough information to enable the user to do simple formatting tasks like making viewgraphs, and to make incremental changes to existing packages of **troff** commands. In most respects, the UNIX formatter **nroff** is identical to **troff**, so this document also serves as a tutorial on **nroff**.

August 4, 1978

†UNIX is a Trademark of Bell Laboratories.

A TROFF Tutorial

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

troff [1] is a text-formatting program, written by J. F. Ossanna, for producing high-quality printed output from the phototypesetter on the UNIX and GCOS operating systems. This document is an example of **troff** output.

The single most important rule of using **troff** is not to use it directly, but through some intermediary. In many ways, **troff** resembles an assembly language — a remarkably powerful and flexible one — but nonetheless such that many operations must be specified at a level of detail and in a form that is too hard for most people to use effectively.

For two special applications, there are programs that provide an interface to **troff** for the majority of users. **eqn** [2] provides an easy to learn language for typesetting mathematics; the **eqn** user need know no **troff** whatsoever to typeset mathematics. **tbl** [3] provides the same convenience for producing tables of arbitrary complexity.

For producing straight text (which may well contain mathematics or tables), there are a number of 'macro packages' that define formatting rules and operations for specific styles of documents, and reduce the amount of direct contact with **troff**. In particular, the '−ms' [4] and PWB/MM [5] packages for Bell Labs internal memoranda and external papers provide most of the facilities needed for a wide range of document preparation. (This memo was prepared with '−ms'.) There are also packages for viewgraphs, for simulating the older **roff** formatters on UNIX and GCOS, and for other special applications. Typically you will find these packages easier to use than **troff** once you get beyond the most trivial operations; you should always consider them first.

In the few cases where existing packages don't do the whole job, the solution is *not* to write an entirely new set of **troff** instructions from scratch, but to make small changes to adapt packages that already exist.

In accordance with this philosophy of letting someone else do the work, the part of **troff** described here is only a small part of the whole, although it tries to concentrate on the more useful parts. In any case, there is no attempt to be complete. Rather, the emphasis is on showing how to do simple things, and how to make incremental changes to what already exists. The contents of the remaining sections are:

2. Point sizes and line spacing
3. Fonts and special characters
4. Indents and line length
5. Tabs
6. Local motions: Drawing lines and characters
7. Strings
8. Introduction to macros
9. Titles, pages and numbering
10. Number registers and arithmetic
11. Macros with arguments
12. Conditionals
13. Environments
14. Diversions
Appendix: Typesetter character set

The **troff** described here is the C-language version running on UNIX at Murray Hill, as documented in [1].

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. (Readers who use **roff** will find the approach familiar.) For **troff** the text and the formatting information are often intertwined quite intimately. Most commands to **troff** are placed on a line separate from the text itself, beginning with a period (one command per line). For example,

```
Some text.
.ps 14
Some more text.
```

will change the 'point size', that is, the size of the letters being printed, to '14 point' (one point is 1/72 inch) like this:

Some text. Some more text.

Occasionally, though, something special occurs in the middle of a line — to produce

Area $= \pi r^2$

you have to type

```
Area = \(*p\fIr\fR\|\s8\u2\d\s0
```

(which we will explain shortly). The backslash character \ is used to introduce **troff** commands and special characters within a line of text.

2. Point Sizes; Line Spacing

As mentioned above, the command .ps 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 ½ inch. There are 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
11 point: Pack my box with five dozen
12 point: Pack my box with five dozen
14 point: Pack my box with five
16 point 18 point 20 point
22 24 28 36

If the number after .ps is not one of these 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 9 point.

The point size can also be changed in the middle of a line or even a word with the in-line command \s. To produce

UNIX runs on a PDP-11/45

type

```
\s8UNIX\s10 runs on a \s8PDP-\s1011/45
```

As above, \s should be followed by a legal point size, except that \s0 causes the size to revert to its previous value. Notice that \s1011 can be understood correctly as 'size 10, followed by an 11', if the size is legal, but not otherwise. Be cautious with similar constructions.

Relative size changes are also legal and useful:

```
\s−2UNIX\s+2
```

temporarily decreases the size, whatever it is, by two points, then restores it. Relative size changes have the advantage that the size difference is independent of the starting size of the document. The amount of the relative change is restricted to a single digit.

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 command to control vertical spacing is .vs. For running text, it is usually best to set the vertical spacing about 20% bigger than the character size. For example, so far in this document, we have used "9 on 11", that is,

```
.ps 9
.vs 11p
```

If we changed to

```
.ps 9
.vs 9p
```

the running text would look like this. After a few lines, you will agree it looks a little cramped. The right vertical spacing is partly a matter of taste, depending on how much text you want to squeeze into a given space, and partly a matter of traditional printing style. By default, **troff** uses 10 on 12.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. This is 12 on 14.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. For example, 10 on 12 uses about twice as much space as 7 on 8. This is 6 on 7, which is even smaller. It packs a lot more words per line, but you can go blind trying to read it.

When used without arguments, .ps and .vs revert to the previous size and vertical spacing respectively.

The command .sp is used to get extra vertical space. Unadorned, it gives you one extra blank line (one .vs, whatever that has been set to). Typically, that's more or less than you want, so .sp can be followed by information about how much space you want —

```
.sp 2i
```

means 'two inches of vertical space'.

```
.sp 2p
```

means 'two points of vertical space'; and

```
.sp 2
```

means 'two vertical spaces' — two of whatever

.vs is set to (this can also be made explicit with .sp 2v); **troff** also understands decimal fractions in most places, so

.sp 1.5i

is a space of 1.5 inches. These same scale factors can be used after .vs to define line spacing, and in fact after most commands 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).

3. Fonts and Special Characters

troff and the typesetter allow four different fonts at any one time. Normally three fonts (Times roman, italic and bold) and one collection of special characters are permanently mounted.

abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

The greek, mathematical symbols and miscellany of the special font are listed in Appendix A.

troff prints in roman unless told otherwise. To switch into bold, use the .ft command

.ft B

and for italics,

.ft I

To return to roman, use .ft **R**; to return to the previous font, whatever it was, use either .ft **P** or just .ft. The 'underline' command

.ul

causes the next input line to print in italics. **.ul** can be followed by a count to indicate that more than one line is to be italicized.

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

bold*face* text

is produced by

\fBbold\fIface\fR text

If you want to do this so the previous font, whatever it was, is left undisturbed, insert extra \fP commands, like this:

\fBbold\fP\fIface\fP\fR text\fP

Because only the immediately previous font is remembered, you have to restore the previous font after each change or you can lose it. The same is true of **.ps** and **.vs** when used without an argument.

There are other fonts available besides the standard set, although you can still use only four at any given time. The command .fp tells **troff** what fonts are physically mounted on the typesetter:

.fp 3 H

says that the Helvetica font is mounted on position 3. (For a complete list of fonts and what they look like, see the **troff** manual.) Appropriate **.fp** commands 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 on 1, italic on 2, bold on 3, and special on 4.

There is also a way to get 'synthetic' bold fonts by overstriking letters with a slight offset. Look at the .bd command in [1].

Special characters have four-character names beginning with \(, and they may be inserted anywhere. For example,

¼ + ½ = ¾

is produced by

\(14 + \(12 = \(34

In particular, greek letters are all of the form \(*−, where − is an upper or lower case roman letter reminiscent of the greek. Thus to get

$\Sigma(\alpha\times\beta) \rightarrow \infty$

in bare **troff** we have to type

\(*S(\(*a\(mu\(*b) \(-> \(if

That line is unscrambled as follows:

\(*S	Σ
(	(
\(*a	α
\(mu	×
\(*b	β
)	)
\(->	→
\(if	∞

A complete list of these special names occurs in Appendix A.

In **eqn** [2] the same effect can be achieved with the input

SIGMA (alpha times beta) −> inf

which is less concise, but clearer to the uninitiated.

Notice that each four-character name is a single character as far as **troff** is concerned — the 'translate' command

.tr \(mi\(em

is perfectly clear, meaning

.tr −—

that is, to translate − into —.

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.

4. Indents and Line Lengths

troff starts with a line length of 6.5 inches, too wide for 8½×11 paper. To reset the line length, use the .ll command, as in

.ll 6i

As with .sp, the actual length can be specified in several ways; inches are probably the most intuitive.

The maximum line length provided by the typesetter is 7.5 inches, by the way. To use the full width, you will have to reset the default physical left margin ("page offset"), which is normally slightly less than one inch from the left edge of the paper. This is done by the .po command.

.po 0

sets the offset as far to the left as it will go.

The indent command **.in** causes the left margin to be indented by some specified amount from the page offset. If we use .in to move the left margin in, and .ll to move the right margin to the left, we can make offset blocks of text:

```
.in 0.3i
.ll −0.3i
text to be set into a block
.ll +0.3i
.in −0.3i
```

will create a block that looks like this:

> Pater noster qui est in caelis sanctificetur nomen tuum; adveniat regnum tuum; fiat voluntas tua, sicut in caelo, et in terra. ... Amen.

Notice the use of '+' and '−' to specify the amount of change. These change the previous setting by the specified amount, rather than just overriding it. The distinction is quite important: .ll +1i makes lines one inch longer; .ll 1i makes them one inch *long*.

With **.in**, **.ll** and **.po**, the previous value is used if no argument is specified.

To indent a single line, use the 'temporary indent' command .ti. For example, all paragraphs in this memo effectively begin with the command

.ti 3

Three of what? The default unit for .ti, as for most horizontally oriented commands (.ll, .in, .po), is ems; an em is roughly the width of the letter 'm' in the current point size. (Precisely, a em in size *p* is *p* points.) 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

causes the next line to be moved back three tenths of an inch. Thus to make a decorative initial capital, we indent the whole paragraph, then move the letter 'P' back with a .ti command:

> **P**ater noster qui est in caelis sanctificetur nomen tuum; adveniat regnum tuum; fiat voluntas tua, sicut in caelo, et in terra. ... Amen.

Of course, there is also some trickery to make the 'P' bigger (just a '\s36P\s0'), and to move it down from its normal position (see the section on local motions).

5. 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, but can be changed by the **.ta** command. To set stops every inch, for example,

```
.ta 1i 2i 3i 4i 5i 6i
```

Unfortunately the stops are left-justified only (as on a typewriter), so lining up columns of right-justified numbers can be painful. If you have many numbers, or if you need more complicated table layout, *don't* use **troff** directly; use the **tbl** program described in [3].

For a handful of numeric columns, you can do it this way: Precede every number by enough blanks to make it line up when typed.

```
.nf
.ta 1i 2i 3i
  1 tab   2 tab   3
 40 tab  50 tab  60
700 tab 800 tab 900
.fi
```

Then change each leading blank into the string \0. This is a character that does not print, but that has the same width as a digit. When printed, this will produce

```
  1        2        3
 40       50       60
700      800      900
```

It is also possible to fill up tabbed-over space with some character other than blanks by setting the 'tab replacement character' with the .tc command:

```
.ta 1.5i 2.5i
.tc \(ru      (\(ru is "_")
Name tab Age tab
```

produces

Name ________________ Age __________

To reset the tab replacement character to a blank, use .tc with no argument. (Lines can also be drawn with the \l command, described in Section 6.)

troff also provides a very general mechanism called 'fields' for setting up complicated columns. (This is used by **tbl**). We will not go into it in this paper.

6. Local Motions: Drawing lines and characters

Remember 'Area = πr^2' and the big 'P' in the Paternoster. How are they done? **troff** provides a host of commands for placing characters of any size at any place. You can use them to draw special characters or to tune your output for a particular appearance. Most of these commands are straightforward, but messy to read and tough to type correctly.

If you won't use **eqn**, subscripts and superscripts are most easily done with the half-line local motions **\u** and **\d**. To go back up the page half a point-size, insert a **\u** at the desired place; to go down, insert a **\d**. (**\u** and **\d** should always be used in pairs, as explained below.) Thus

```
Area = \(*pr\u2\d
```

produces

Area = πr^2

To make the '2' smaller, bracket it with **\s−2**...**\s0**. Since **\u** and **\d** refer to the current point size, be sure to put them either both inside or both outside the size changes, or you will get an unbalanced vertical motion.

Sometimes the space given by **\u** and **\d** isn't the right amount. The **\v** command can be used to request an arbitrary amount of vertical motion. The in-line command

```
\v'(amount)'
```

causes motion up or down the page by the amount specified in '(amount)'. For example, to move the 'P' down, we used

```
.in +0.6i         (move paragraph in)
.ll −0.3i         (shorten lines)
.ti −0.3i         (move P back)
\v'2'\s36P\s0\v'−2'ater noster qui est
in caelis ...
```

A minus sign causes upward motion, while no sign or a plus sign means down the page. Thus \v'−2' causes an upward vertical motion of two line spaces.

There are many other ways to specify the amount of motion —

```
\v'0.1i'
\v'3p'
\v'−0.5m'
```

and so on are all legal. Notice that the scale specifier **i** or **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.

Arbitrary horizontal motions are also available — **\h** is quite analogous to **\v**, except that the default scale factor is ems instead of line spaces. As an example,

```
\h'−0.1i'
```

causes a backwards motion of a tenth of an inch. As a practical matter, consider printing the mathematical symbol '>>'. The default spacing is too wide, so **eqn** replaces this by

```
>\h'−0.3m'>
```

to produce >>.

Frequently \h is used with the 'width function' \w to generate motions equal to the width of some character string. The construction

```
\w'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, all of the command names in the text, like **.sp**, were done by overstriking with a slight offset. The commands for **.sp** are

```
.sp\h'−\w'.sp'u'\h'1u'.sp
```

That is, put out '.sp', move left by the width of '.sp', move right 1 unit, and print '.sp' again. (Of course there is a way to avoid typing that much input for each command name, which we will discuss in Section 11.)

There are also several special-purpose **troff** commands for local motion. We have already seen \0, which is an unpaddable white space of the same width as a digit. 'Unpaddable' means that it will never be widened or split across a line by line justification and filling. There is also \(blank), which is an unpaddable character the width of a space, \|, which is half that width, \^, which is one quarter of the width of a space, and \&, which has zero width. (This last one is useful, for example, in entering a text line which would otherwise begin with a '.'.)

The command \o, used like

```
\o'set of characters'
```

causes (up to 9) characters to be overstruck, centered on the widest. This is nice for accents, as in

```
syst\o"e\(ga"me t\o"e\(aa"l\o"e\(aa"phonique
```

which makes

système téléphonique

The accents are **\(ga** and **\(aa**, or \` and \´; remember that each is just one character to **troff**.

You can make your own overstrikes with another special convention, **\z**, the zero-motion command. **\zx** suppresses the normal horizontal motion after printing the single character **x**, so another character can be laid on top of it. Although sizes can be changed within \o, it centers the characters on the widest, and there can be no horizontal or vertical motions, so \z may be the only way to get what you want:

□

is produced by

```
.sp 2
\s8\z\(sq\s14\z\(sq\s22\z\(sq\s36\(sq
```

The **.sp** is needed to leave room 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
```

'0.25m' is an empirical constant.

A more ornate overstrike is given by the bracketing function **\b**, which piles up characters vertically, centered on the current baseline. Thus we can get big brackets, constructing them with piled-up smaller pieces:

{[x]}

by typing in only this:

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

troff also provides a convenient facility for drawing horizontal and vertical lines of arbitrary length with arbitrary characters. **\l'1i'** draws a line one inch long, like this: ________________. The length can be followed by the character to use if the _ isn't appropriate; **\l'0.5i.'** draws a half-inch line of dots: .... The construction **\L** is entirely analogous, except that it draws a vertical line instead of horizontal.

7. 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.

Fortunately, **troff** provides a way in which you can store an arbitrary collection of text in a 'string', and thereafter use the string name as a shorthand for its contents. Strings are one of several **troff** mechanisms whose judicious use lets you type a document with less effort and organize it so that extensive format changes can be made with few editing changes.

A reference to a string is replaced by whatever text the string was defined as. Strings are defined with the command **.ds**. The line

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

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

String names may be either one or two characters long, and are referred to by ***x** for one character names or ***(xy** for two character names. Thus to get téléphone, given the definition of the string e as above, we can say t*el*ephone.

If a string must begin with blanks, define it as

```
.ds xx "     text
```

The double quote signals the beginning of the definition. There is no trailing quote; the end of the line terminates the string.

A string may actually be several lines long; if **troff** encounters a \ at the end of *any* line, it is thrown away and the next line added to the current one. So you can make a long string simply by ending each line but the last with a backslash:

```
.ds xx this \
is a very \
long string
```

Strings may be defined in terms of other strings, or even in terms of themselves; we will discuss some of these possibilities later.

8. Introduction to Macros

Before we can go much further in **troff**, we need to learn a bit about the macro facility. In its simplest form, a macro is just a shorthand notation quite similar to a string. Suppose we want every paragraph to start in exactly the same way — with a space and a temporary indent of two ems:

```
.sp
.ti +2m
```

Then to save typing, we would like to collapse these into one shorthand line, a **troff** 'command' like

```
.PP
```

that would be treated by **troff** exactly as

```
.sp
.ti +2m
```

.PP is called a *macro*. The way we tell **troff** what **.PP** means is to *define* it with the **.de** command:

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

The first line names the macro (we used '**.PP**' for 'paragraph', and upper case so it wouldn't conflict with any name that **troff** might already know about). The last line .. marks the end of the definition. In between is the text, which is simply inserted whenever **troff** sees the 'command' or macro call

```
.PP
```

A macro can contain any mixture of text and formatting commands.

The definition of **.PP** has to precede its first use; undefined macros are simply ignored. Names are restricted to one or two characters.

Using macros for commonly occurring sequences of commands is critically important. Not only does it save typing, but it makes later changes much easier. Suppose we decide that the paragraph indent is too small, the vertical space is much too big, and roman font should be forced. Instead of changing the whole document, we need only change the definition of **.PP** to something like

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

and the change takes effect everywhere we used **.PP**.

\\" is a **troff** command that causes the rest of the line to be ignored. We use it here to add comments to the macro definition (a wise idea once definitions get complicated).

As another example of macros, consider these two which start and end a block of offset, unfilled text, like most of the examples in this paper:

```
.de BS        \" start indented block
.sp
.nf
.in +0.3i
..
.de BE        \" end indented block
.sp
.fi
.in −0.3i
..
```

Now we can surround text like

```
Copy to
John Doe
Richard Roberts
Stanley Smith
```

by the commands **.BS** and **.BE**, and it will come out as it did above. Notice that we indented by **.in +0.3i** instead of **.in 0.3i**. This way we can nest our uses of **.BS** and **BE** to get blocks within blocks.

If later on we decide that the indent should be 0.5i, then it is only necessary to change the definitions of **.BS** and **.BE**, not the whole paper.

9. Titles, Pages and Numbering

This is an area where things get tougher, because nothing is done for you automatically. Of necessity, some of this section is a cookbook, to be copied literally until you get some experience.

Suppose you want a title at the top of each page, saying just

‾‾‾‾left top   center top   right top‾‾‾‾

In **roff**, one can 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**, a serious hardship for the novice. Instead you have to do a lot of specification.

You have to say what the actual title is (easy); when to print it (easy enough); and what to do at and around the title line (harder). Taking these in reverse order, first we define a macro **.NP** (for 'new page') to process titles and the like at the end of one page and the beginning of the next:

```
.de NP
'bp
'sp 0.5i
.tl 'left top'center top'right top'
'sp 0.3i
..
```

To make sure we're at the top of a page, we issue a 'begin page' command **'bp**, which causes a skip to top-of-page (we'll explain the **'** shortly). Then we space down half an inch, print the title (the use of **.tl** should be self explanatory; later we will discuss parameterizing the titles), space another 0.3 inches, and we're done.

To ask for **.NP** at the bottom of each page, we have to say something like 'when the text is within an inch of the bottom of the page, start the processing for a new page.' This is done with a 'when' command **.wh**:

```
.wh −1i NP
```

(No '.' is used before NP; this is simply the name of a macro, not a macro call.) The minus sign means 'measure up from the bottom of the page', so '−1i' means 'one inch from the bottom'.

The **.wh** command appears in the input outside the definition of **.NP**; typically the input would be

```
.de NP
...
..
.wh −1i NP
```

Now what happens? As text is actually being output, **troff** keeps track of its vertical position on the page, and after a line is printed within one inch from the bottom, the **.NP** macro is activated. (In the jargon, the **.wh** command sets a *trap* at the specified place, which is 'sprung' when that point is passed.) **.NP** causes a skip to the top of the next page (that's what the **'bp** was for), then prints the title with the appropriate margins.

Why **'bp** and **'sp** instead of **.bp** and **.sp**? The answer is that **.sp** and **.bp**, like several other commands, cause a *break* to take place. That is, all the input text collected but not yet printed is flushed out as soon as possible, and the next input line is guaranteed to start a new line of output. If we had used **.sp** or **.bp** in the **.NP** macro, this would cause a break 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. This is *not* what we want. Using **'** instead of **.** for a command tells **troff** that no break is to take place — the output line currently being filled should *not* be forced out before the space or new page.

The list of commands that cause a break is short and natural:

```
.bp  .br  .ce  .fi  .nf  .sp  .in  .ti
```

All others cause *no* break, regardless of whether

you use a . or a '. If you really need a break, add a **.br** command at the appropriate place.

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** command.

There are several ways to fix the problems of point sizes and fonts in titles. For the simplest applications, we can change **.NP** to set the proper size and font for the title, then restore the previous values, like this:

```
.de NP
'bp
'sp 0.5i
.ft R          \" set title font to roman
.ps 10         \" and size to 10 point
.lt 6i         \" and length to 6 inches
.tl 'left'center'right'
.ps            \" revert to previous size
.ft P          \" and to previous font
'sp 0.3i
..
```

This version of **.NP** does *not* work if the fields in the .tl command contain size or font changes. To cope with that requires **troff**'s 'environment' mechanism, which we will discuss in Section 13.

To get a footer at the bottom of a page, you can modify **.NP** so it does some processing before the **'bp** command, or split the job into a footer macro invoked at the bottom margin and a header macro invoked at the top of the page. These variations are left as exercises.

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, as on this page. You can set the page number at any time with either **.bp n**, which immediately starts a new page numbered n, or with **.pn n**, which sets the page number for the next page but doesn't cause a skip to the new page. Again, **.bp +n** sets the page number to **n** more than its current value; **.bp** means **.bp +1**.

10. Number Registers and Arithmetic

troff has a facility for doing arithmetic, and for defining and using variables with numeric values, called *number registers.* Number registers, like strings and macros, can be useful in setting up a document so it is easy to change later. And of course they serve for any sort of arithmetic computation.

Like strings, number registers have one or two character names. They are set by the **.nr** command, and are referenced anywhere by **\nx** (one character name) or **\n(xy** (two character name).

There are quite a few pre-defined number registers maintained by **troff**, among them % for the current page number; **nl** for the current vertical position on the page; **dy**, **mo** and **yr** for the current day, month and year; and **.s** and **.f** for the current size and font. (The font is a number from 1 to 4.) Any of these can be used in computations like any other register, but some, like **.s** and **.f**, cannot be changed with **.nr**.

As an example of the use of number registers, in the **−ms** macro package [4], most significant parameters are defined in terms of the values of a handful of number registers. These include the point size for text, the vertical spacing, and the line and title lengths. To set the point size and vertical spacing for the following paragraphs, for example, a user may say

```
.nr PS 9
.nr VS 11
```

The paragraph macro **.PP** is defined (roughly) as follows:

```
.de PP
.ps \\n(PS        \" reset size
.vs \\n(VSp       \" spacing
.ft R             \" font
.sp 0.5v          \" half a line
.ti +3m
..
```

This sets the font to Roman and the point size and line spacing to whatever values are stored in the number registers PS and VS.

Why are there two backslashes? This is the eternal problem of how to quote a quote. When **troff** originally reads the macro definition, it peels off 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 in 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 it is used.

Protecting by an extra layer of backslashes

is only needed for \n, *, \$ (which we haven't come to yet), and \ itself. Things like \s, \f, \h, \v, and so on do not need an extra backslash, since they are converted by **troff** to an internal code immediately upon being seen.

Arithmetic expressions can appear anywhere that a number is expected. As a trivial example,

```
.nr PS \\n(PS-2
```

decrements PS by 2. Expressions can use the arithmetic operators +, −, *, /, % (mod), the relational operators >, >=, <, <=, =, and != (not equal), and parentheses.

Although the arithmetic we have done so far has been straightforward, more complicated things are somewhat tricky. First, number registers hold only integers. **troff** arithmetic uses truncating integer division, just like Fortran. Second, in the absence of parentheses, evaluation is done 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 when you wouldn't expect it — in particular, when arithmetic is being done in a context that implies horizontal or vertical dimensions. For example,

```
.ll 7/2i
```

would seem obvious enough — 3½ inches. Sorry. Remember that the default units for horizontal parameters like **.ll** are ems. That's really '7 ems / 2 inches', and when translated into machine units, it becomes zero. How about

```
.ll 7i/2
```

Sorry, 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** command, there is no implication of horizontal or vertical dimension, so the default units are 'units', and 7i/2 and 7i/2u mean the same thing. Thus

```
.nr ll 7i/2
.ll \\n(llu
```

does just what you want, so long as you don't forget the **u** on the **.ll** command.

11. 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 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.

Let us illustrate by defining a macro **.SM** that will print its argument two points smaller than the surrounding text. That is, the macro call

```
.SM TROFF
```

will produce TROFF.

The definition of **.SM** is

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

Within a macro definition, the symbol **\\$n** refers to the **n**th argument that the macro was called with. Thus **\\$1** is the string to be placed in a smaller point size when **.SM** is called.

As a slightly more complicated version, the following definition of **.SM** permits optional second and third arguments that will be printed in the normal size:

```
.de SM
\\$3\s-2\\$1\s+2\\$2
..
```

Arguments not provided when the macro is called are treated as empty, so

```
.SM TROFF ),
```

produces TROFF), while

```
.SM TROFF ). (
```

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

By the way, the number of arguments that a macro was called with is available in number register **.$**.

The following macro **.BD** is the one used to make the 'bold roman' we have been using for **troff** command names in text. It combines horizontal motions, width computations, and argument rearrangement.

```
.de BD
\&\\$3\f1\\$1\h'−\w'\\$1'u+1u'\\$1\fP\\$2
..
```

The \h and \w commands need no extra backslash, as we discussed above. The \& is there in case the argument begins with a period.

Two backslashes are needed with the \\$n commands, though, to protect one of them when the macro is being defined. Perhaps a second example will make this clearer. Consider a macro called .SH which produces section headings rather like those in this paper, with the sections numbered automatically, and the title in bold in a smaller size. The use is

```
.SH  "Section title ..."
```

(If the argument to a macro is to contain blanks, then it must be *surrounded* by double quotes, unlike a string, where only one leading quote is permitted.)

Here is the definition of the .SH macro:

```
.nr SH 0        \" initialize section number
.de SH
.sp 0.3i
.ft B
.nr SH \\n(SH+1   \" increment number
.ps \\n(PS−1      \" decrease PS
\\n(SH. \\$1      \" number. title
.ps \\n(PS        \" restore PS
.sp 0.3i
.ft R
..
```

The section number is kept in number register SH, which is incremented each time just before it is used. (A number register may have the same name as a macro without conflict but a string may not.)

We used \\n(SH instead of \n(SH and \\n(PS instead of \n(PS. If we had used \n(SH, we would get the value of the register at the time the macro was *defined*, not at the time it was *used*. If that's what you want, fine, but not here. Similarly, by using \\n(PS, we get the point size at the time the macro is called.

As an example that does not involve numbers, recall our .NP macro which had a

```
.tl 'left'center'right'
```

We could make these into parameters by using instead

```
.tl '\\*(LT'\\*(CT'\\*(RT'
```

so the title comes from three strings called LT, CT and RT. If these are empty, then the title will be a blank line. Normally CT would be set with something like

```
.ds CT - % -
```

to give just the page number between hyphens (as on the top of this page), but a user could supply private definitions for any of the strings.

12. Conditionals

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 command 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 command. If the condition is false, or zero or negative, the rest of the line is skipped.

It is possible to do more than one command if a condition is true. Suppose several operations are to be done before section 1. One possibility is to define a macro .S1 and invoke it if we are about to do section 1 (as determined by an .if).

```
.de S1
--- processing for section 1 ---
..
.de SH
...
.if \\n(SH=1 .S1
...
..
```

An alternate way is to use the extended form of the .if, like this:

```
.if \\n(SH=1 \{--- processing
for section 1 ----\}
```

The braces \{ and \} must occur in the positions shown or you will get unexpected extra lines in your output. **troff** also provides an 'if-else' construction, which 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.

13. Environments

As we mentioned, there is a potential problem when going across a page boundary: parameters like size and font for a page title may well be different from those in effect in the text when the page boundary occurs. **troff** provides a very general way to deal with this and similar situations. There are three 'environments', each of which has independently settable versions of many of the parameters associated with processing, including size, font, line and title lengths, fill/nofill mode, tab stops, and even partially collected lines. Thus the titling problem may be readily solved by processing the main text in one environment and titles in a separate one with its own suitable parameters.

The command .ev **n** shifts to environment **n**; **n** must be 0, 1 or 2. The command **.ev** with no argument returns to the previous environment. Environment names are maintained in a stack, so calls for different environments may be nested and unwound consistently.

Suppose we say that the main text is processed in environment 0, which is where **troff** begins by default. Then we can modify the new page macro .**NP** to process titles in environment 1 like this:

```
.de NP
.ev 1          \" shift to new environment
.lt 6i         \" set parameters here
.ft R
.ps 10
... any other processing ...
.ev            \" return to previous environment
..
```

It is also possible to initialize the parameters for an environment outside the .**NP** macro, but the version shown keeps all the processing in one place and is thus easier to understand and change.

14. Diversions

There are numerous occasions in page layout when it is necessary to store some text for a period of time without actually printing it. Footnotes are the most obvious example: the text of the footnote usually appears in the input well before the place on the page where it is to be printed is reached. In fact, the place where it is output normally depends on how big it is, which implies that there must be a way to process the footnote at least enough to decide its size without printing it.

troff provides a mechanism called a diversion for doing this processing. Any part of the output may be diverted into a macro instead of being printed, and then at some convenient time the macro may be put back into the input.

The command **.di xy** begins a diversion — all subsequent output is collected into the macro **xy** until the command **.di** with no arguments is encountered. This terminates the diversion. The processed text is available at any time thereafter, simply by giving the command

```
.xy
```

The vertical size of the last finished diversion is contained in the built-in number register **dn**.

As a simple example, suppose we want to implement a 'keep-release' operation, so that text between the commands .KS and .KE will not be split across a page boundary (as for a figure or table). Clearly, when a .KS is encountered, we have to begin diverting the output so we can find out how big it is. Then when a .KE is seen, we decide whether the diverted text will fit on the current page, and print it either there if it fits, or at the top of the next page if it doesn't. So:

```
.de KS      \" start keep
.br         \" start fresh line
.ev 1       \" collect in new environment
.fi         \" make it filled text
.di XX      \" collect in XX
..

.de KE      \" end keep
.br         \" get last partial line
.di         \" end diversion
.if \\n(dn>=\\n(.t .bp  \" bp if doesn't fit
.nf         \" bring it back in no-fill
.XX         \" text
.ev         \" return to normal environment
..
```

Recall that number register **nl** is the current

position on the output page. Since output was being diverted, this remains at its value when the diversion started. **dn** is the amount of text in the diversion; .t (another built-in register) is the distance to the next trap, which we assume is at the bottom margin of the page. If the diversion is large enough to go past the trap, the **.if** is satisfied, and a **.bp** is issued. In either case, the diverted output is then brought back with **.XX**. It is essential to bring it back in no-fill mode so **troff** will do no further processing on it.

This is not the most general keep-release, nor is it robust in the face of all conceivable inputs, but it would require more space than we have here to write it in full generality. This section is not intended to teach everything about diversions, but to sketch out enough that you can read existing macro packages with some comprehension.

Acknowledgements

I am deeply indebted to J. F. Ossanna, the author of **troff**, for his repeated patient explanations of fine points, and for his continuing willingness to adapt **troff** to make other uses easier. I am also grateful to Jim Blinn, Ted Dolotta, Doug McIlroy, Mike Lesk and Joel Sturman for helpful comments on this paper.

References

[1] J. F. Ossanna, *NROFF/TROFF* User's Manual, Bell Laboratories Computing Science Technical Report 54, 1976.

[2] B. W. Kernighan, *A System for Typesetting Mathematics — User's Guide (Second Edition)*, Bell Laboratories Computing Science Technical Report 17, 1977.

[3] M. E. Lesk, *TBL — A Program to Format Tables*, Bell Laboratories Computing Science Technical Report 49, 1976.

[4] M. E. Lesk, *Typing Documents on UNIX*, Bell Laboratories, 1978.

[5] J. R. Mashey and D. W. Smith, *PWB/MM — Programmer's Workbench Memorandum Macros*, Bell Laboratories internal memorandum.

Appendix A: Phototypesetter Character Set

These characters exist in roman, italic, and bold. To get the one on the left, type the four-character name on the right.

ﬀ	\(ff	ﬁ	\(fi	ﬂ	\(fl	ﬃ	\(Fi	ﬄ	\(Fl
_	\(ru	—	\(em	¼	\(14	½	\(12	¾	\(34
©	\(co	°	\(de	†	\(dg	′	\(fm	¢	\(ct
®	\(rg	•	\(bu	□	\(sq	-	\(hy		

(In bold, \(sq is ■.)

The following are special-font characters:

+	\(pl	−	\(mi	×	\(mu	÷	\(di
=	\(eq	≡	\(==	≥	\(>=	≤	\(<=
≠	\(!=	±	\(+-	¬	\(no	/	\(sl
~	\(ap	≃	\(~=	∝	\(pt	∇	\(gr
→	\(->	←	\(<-	↑	\(ua	↓	\(da
∫	\(is	∂	\(pd	∞	\(if	√	\(sr
⊂	\(sb	⊃	\(sp	∪	\(cu	∩	\(ca
⊆	\(ib	⊇	\(ip	∈	\(mo	∅	\(es
´	\(aa	‵	\(ga	○	\(ci		\(bs
§	\(sc	‡	\(dd	☜	\(lh	☞	\(rh
⎧	\(lt	⎫	\(rt	⎡	\(lc	⎤	\(rc
⎩	\(lb	⎭	\(rb	⎣	\(lf	⎦	\(rf
⎨	\(lk	⎬	\(rk	│	\(bv	ς	\(ts
│	\(br	∣	\(or	_	\(ul	‾	\(rn
*	\(**						

These four characters also have two-character names. The ´ is the apostrophe on terminals; the ‵ is the other quote mark.

´ \'   ‵ \‵   − \-   _ _

These characters exist only on the special font, but they do not have four-character names:

" { } < > ˜ ˆ \ # @

For greek, precede the roman letter by \(* to get the corresponding greek; for example, \(*a is α.

a b g d e z y h i k l m n c o p r s t u f x q w

α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ σ τ υ φ χ ψ ω

A B G D E Z Y H I K L M N C O P R S T U F X Q W

Α Β Γ Δ Ε Ζ Η Θ Ι Κ Λ Μ Ν Ξ Ο Π Ρ Σ Τ Υ Φ Χ Ψ Ω

Tbl — A Program to Format Tables

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Tbl is a document formatting preprocessor for *troff* or *nroff* which makes even fairly complex tables easy to specify and enter. It is available on the PDP-11 UNIX* system and on Honeywell 6000 GCOS. Tables are made up of columns which may be independently centered, right-adjusted, left-adjusted, or aligned by decimal points. Headings may be placed over single columns or groups of columns. A table entry may contain equations, or may consist of several rows of text. Horizontal or vertical lines may be drawn as desired in the table, and any table or element may be enclosed in a box. For example:

1970 Federal Budget Transfers (in billions of dollars)			
State	Taxes collected	Money spent	Net
New York	22.91	21.35	−1.56
New Jersey	8.33	6.96	−1.37
Connecticut	4.12	3.10	−1.02
Maine	0.74	0.67	−0.07
California	22.29	22.42	+0.13
New Mexico	0.70	1.49	+0.79
Georgia	3.30	4.28	+0.98
Mississippi	1.15	2.32	+1.17
Texas	9.33	11.13	+1.80

January 16, 1979

* UNIX is a Trademark/Service Mark of the Bell System

Tbl — A Program to Format Tables

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction.

Tbl turns a simple description of a table into a *troff* or *nroff* [1] program (list of commands) that prints the table. *Tbl* may be used on the PDP-11 UNIX [2] system and on the Honeywell 6000 GCOS system. It attempts to isolate a portion of a job that it can successfully handle and leave the remainder for other programs. Thus *tbl* may be used with the equation formatting program *eqn* [3] or various layout macro packages [4,5,6], but does not duplicate their functions.

This memorandum is divided into two parts. First we give the rules for preparing *tbl* input; then some examples are shown. The description of rules is precise but technical, and the beginning user may prefer to read the examples first, as they show some common table arrangements. A section explaining how to invoke *tbl* precedes the examples. To avoid repetition, henceforth read *troff* as "*troff* or *nroff*."

The input to *tbl* is text for a document, with tables preceded by a ".TS" (table start) command and followed by a ".TE" (table end) command. *Tbl* processes the tables, generating *troff* formatting commands, and leaves the remainder of the text unchanged. The ".TS" and ".TE" lines are copied, too, so that *troff* page layout macros (such as the memo formatting macros [4]) can use these lines to delimit and place tables as they see fit. In particular, any arguments on the ".TS" or ".TE" lines are copied but otherwise ignored, and may be used by document layout macro commands.

The format of the input is as follows:

text
.TS
table
.TE
text
.TS
table
.TE
text
. . .

where the format of each table is as follows:

.TS
options ;
format .
data
.TE

Each table is independent, and must contain formatting information followed by the data to be entered in the table. The formatting information, which describes the individual columns and rows of the table, may be preceded by a few options that affect the entire table. A detailed description of tables is given in the next section.

Input commands.

As indicated above, a table contains, first, global options, then a format section describing the layout of the table entries, and then the data to be printed. The format and data are always required, but not the options. The various parts of the table are entered as follows:

1) OPTIONS. There may be a single line of options affecting the whole table. If present, this line must follow the .TS line immediately and must contain a list of option names separated by spaces, tabs, or commas, and must be terminated by a semicolon. The allowable options are:

   center — center the table (default is left-adjust);

   expand — make the table as wide as the current line length;

   box — enclose the table in a box;

   allbox — enclose each item in the table in a box;

   doublebox — enclose the table in two boxes;

   tab (*x*) — use *x* instead of tab to separate data items.

   linesize (*n*) — set lines or rules (e.g. from **box**) in *n* point type;

   delim (*xy*) — recognize *x* and *y* as the *eqn* delimiters.

   The *tbl* program tries to keep boxed tables on one page by issuing appropriate "need" (.*ne*) commands. These requests are calculated from the number of lines in the tables, and if there are spacing commands embedded in the input, these requests may be inaccurate; use normal *troff* procedures, such as keep-release macros, in that case. The user who must have a multi-page boxed table should use macros designed for this purpose, as explained below under 'Usage.'

2) FORMAT. The format section of the table specifies the layout of the columns. Each line in this section corresponds to one line of the table (except that the last line corresponds to all following lines up to the next .T&, if any — see below), and each line contains a key-letter for each column of the table. It is good practice to separate the key letters for each column by spaces or tabs. Each key-letter is one of the following:

   L or **l** to indicate a left-adjusted column entry;

   R or **r** to indicate a right-adjusted column entry;

   C or **c** to indicate a centered column entry;

   N or **n** to indicate a numerical column entry, to be aligned with other numerical entries so that the units digits of numbers line up;

   A or **a** to indicate an alphabetic subcolumn; all corresponding entries are aligned on the left, and positioned so that the widest is centered within the column (see example on page 12);

   S or **s** to indicate a spanned heading, i.e. to indicate that the entry from the previous column continues across this column (not allowed for the first column, obviously); or

   ^ to indicate a vertically spanned heading, i.e. to indicate that the entry from the previous row continues down through this row. (Not allowed for the first row of the table, obviously).

   When numerical alignment is specified, a location for the decimal point is sought. The rightmost dot (.) adjacent to a digit is used as a decimal point; if there is no dot adjoining a digit, the rightmost digit is used as a units digit; if no alignment is indicated, the item is centered in the column. However, the special non-printing character string \& may be used to override unconditionally dots and digits, or to align alphabetic data; this string lines up where a dot normally would, and then disappears from the final output. In the example below, the items shown at the left will be aligned (in a numerical column) as

shown on the right:

```
13          13
4.2          4.2
26.4.12     26.4.12
abc           abc
abc\&        abc
43\&3.22     433.22
749.12      749.12
```

Note: If numerical data are used in the same column with wider **L** or **r** type table entries, the widest *number* is centered relative to the wider **L** or **r** items (**L** is used instead of **l** for readability; they have the same meaning as key-letters). Alignment within the numerical items is preserved. This is similar to the behavior of **a** type data, as explained above. However, alphabetic subcolumns (requested by the **a** key-letter) are always slightly indented relative to **L** items; if necessary, the column width is increased to force this. This is not true for **n** type entries.

Warning: the **n** and **a** items should not be used in the same column.

For readability, the key-letters describing each column should be separated by spaces. The end of the format section is indicated by a period. The layout of the key-letters in the format section resembles the layout of the actual data in the table. Thus a simple format might appear as:

```
c s s
l n n .
```

which specifies a table of three columns. The first line of the table contains a heading centered across all three columns; each remaining line contains a left-adjusted item in the first column followed by two columns of numerical data. A sample table in this format might be:

Overall title		
Item-a	34.22	9.1
Item-b	12.65	.02
Items: c,d,e	23	5.8
Total	69.87	14.92

There are some additional features of the key-letter system:

Horizontal lines — A key-letter may be replaced by '_' (underscore) to indicate a horizontal line in place of the corresponding column entry, or by '=' to indicate a double horizontal line. If an adjacent column contains a horizontal line, or if there are vertical lines adjoining this column, this horizontal line is extended to meet the nearby lines. If any data entry is provided for this column, it is ignored and a warning message is printed.

Vertical lines — A vertical bar may be placed between column key-letters. This will cause a vertical line between the corresponding columns of the table. A vertical bar to the left of the first key-letter or to the right of the last one produces a line at the edge of the table. If two vertical bars appear between key-letters, a double vertical line is drawn.

Space between columns — A number may follow the key-letter. This indicates the amount of separation between this column and the next column. The number normally specifies the separation in *ens* (one en is about the width of the letter 'n').* If the "expand" option is used, then these numbers are multiplied by a constant such that the table is as wide as the current line length. The default column separation

* More precisely, an en is a number of points (1 point = 1/72 inch) equal to half the current type size.

number is 3. If the separation is changed the worst case (largest space requested) governs.

Vertical spanning — Normally, vertically spanned items extending over several rows of the table are centered in their vertical range. If a key-letter is followed by **t** or **T**, any corresponding vertically spanned item will begin at the top line of its range.

Font changes — A key-letter may be followed by a string containing a font name or number preceded by the letter **f** or **F**. This indicates that the corresponding column should be in a different font from the default font (usually Roman). All font names are one or two letters; a one-letter font name should be separated from whatever follows by a space or tab. The single letters **B**, **b**, **I**, and **i** are shorter synonyms for **fB** and **fI**. Font change commands given with the table entries override these specifications.

Point size changes — A key-letter may be followed by the letter **p** or **P** and a number to indicate the point size of the corresponding table entries. The number may be a signed digit, in which case it is taken as an increment or decrement from the current point size. If both a point size and a column separation value are given, one or more blanks must separate them.

Vertical spacing changes — A key-letter may be followed by the letter **v** or **V** and a number to indicate the vertical line spacing to be used within a multi-line corresponding table entry. The number may be a signed digit, in which case it is taken as an increment or decrement from the current vertical spacing. A column separation value must be separated by blanks or some other specification from a vertical spacing request. This request has no effect unless the corresponding table entry is a text block (see below).

Column width indication — A key-letter may be followed by the letter **w** or **W** and a width value in parentheses. This width is used as a minimum column width. If the largest element in the column is not as wide as the width value given after the **w**, the largest element is assumed to be that wide. If the largest element in the column is wider than the specified value, its width is used. The width is also used as a default line length for included text blocks. Normal *troff* units can be used to scale the width value; if none are used, the default is ens. If the width specification is a unitless integer the parentheses may be omitted. If the width value is changed in a column, the *last* one given controls.

Equal width columns — A key-letter may be followed by the letter **e** or **E** to indicate equal width columns. All columns whose key-letters are followed by **e** or **E** are made the same width. This permits the user to get a group of regularly spaced columns.

Note: The order of the above features is immaterial; they need not be separated by spaces, except as indicated above to avoid ambiguities involving point size and font changes. Thus a numerical column entry in italic font and 12 point type with a minimum width of 2.5 inches and separated by 6 ens from the next column could be specified as

```
np12w(2.5i)fI 6
```

Alternative notation — Instead of listing the format of successive lines of a table on consecutive lines of the format section, successive line formats may be given on the same line, separated by commas, so that the format for the example above might have been written:

```
c s s, l n n .
```

Default — Column descriptors missing from the end of a format line are assumed to be L. The longest line in the format section, however, defines the number of columns in the table; extra columns in the data are ignored silently.

3) DATA. The data for the table are typed after the format. Normally, each table line is typed as one line of data. Very long input lines can be broken: any line whose last character is \ is combined with the following line (and the \ vanishes). The data for different columns (the table entries) are separated by tabs, or by whatever character has been specified in the option *tabs* option. There are a few special cases:

Troff commands within tables — An input line beginning with a '.' followed by anything but a number is assumed to be a command to *troff* and is passed through unchanged, retaining its position in the table. So, for example, space within a table may be produced by ".sp" commands in the data.

Full width horizontal lines — An input *line* containing only the character _ (underscore) or = (equal sign) is taken to be a single or double line, respectively, extending the full width of the *table*.

Single column horizontal lines — An input table *entry* containing only the character _ or = is taken to be a single or double line extending the full width of the *column*. Such lines are extended to meet horizontal or vertical lines adjoining this column. To obtain these characters explicitly in a column, either precede them by \& or follow them by a space before the usual tab or newline.

Short horizontal lines — An input table *entry* containing only the string _ is taken to be a single line as wide as the contents of the column. It is not extended to meet adjoining lines.

Repeated characters — An input table *entry* containing only a string of the form \R*x* where *x* is any character is replaced by repetitions of the character *x* as wide as the data in the column. The sequence of *x*'s is not extended to meet adjoining columns.

Vertically spanned items — An input table entry containing only the character string \^ indicates that the table entry immediately above spans downward over this row. It is equivalent to a table format key-letter of '^'.

Text blocks — In order to include a block of text as a table entry, precede it by **T{** and follow it by **T}**. Thus the sequence

. . . **T{**
block of
text
T} . . .

is the way to enter, as a single entry in the table, something that cannot conveniently be typed as a simple string between tabs. Note that the **T}** end delimiter must begin a line; additional columns of data may follow after a tab on the same line. See the example on page 10 for an illustration of included text blocks in a table. If more than twenty or thirty text blocks are used in a table, various limits in the *troff* program are likely to be exceeded, producing diagnostics such as 'too many string/macro names' or 'too many number registers.'

Text blocks are pulled out from the table, processed separately by *troff*, and replaced in the table as a solid block. If no line length is specified in the *block of text* itself, or in the table format, the default is to use $L \times C/(N+1)$ where L is the current line length, C is the number of table columns spanned by the text, and N is the total number of columns in the table. The other parameters (point size, font, etc.) used in setting the *block of text* are those in effect at the beginning of the table (including the effect of the ".TS" macro) and any table format specifications of size, spacing and font, using the **p**, **v** and **f** modifiers to the column key-letters. Commands within the text block itself are also recognized, of course. However, *troff* commands within the table data but not within the text block do not affect that block.

Warnings: — Although any number of lines may be present in a table, only the first 200 lines are used in calculating the widths of the various columns. A multi-page table, of course, may be arranged as several single-page tables if this proves to be a problem. Other difficulties with formatting may arise because, in the calculation of column widths all table entries are assumed to be in the font and size being used when the ".TS" command was encountered, except for font and size changes indicated (a) in the table format section and (b) within the table data (as in the entry \s+3\fIdata\fP\s0). Therefore, although arbitrary *troff* requests may be sprinkled in a table, care must be taken to avoid confusing the width calculations; use requests such as '.ps' with care.

4) ADDITIONAL COMMAND LINES. If the format of a table must be changed after many similar lines, as with sub-headings or summarizations, the ".T&" (table continue) command can be used to change column parameters. The outline of such a table input is:

```
.TS
options ;
format .
data
. . .
.T&
format .
data
.T&
format .
data
.TE
```

as in the examples on pages 10 and 12. Using this procedure, each table line can be close to its corresponding format line.

Warning: it is not possible to change the number of columns, the space between columns, the global options such as *box*, or the selection of columns to be made equal width.

Usage.

On UNIX, *tbl* can be run on a simple table with the command

tbl input-file | troff

but for more complicated use, where there are several input files, and they contain equations and *ms* memorandum layout commands as well as tables, the normal command would be

tbl file-1 file-2 . . . | eqn | troff −ms

and, of course, the usual options may be used on the *troff* and *eqn* commands. The usage for *nroff* is similar to that for *troff*, but only TELETYPE® Model 37 and Diablo-mechanism (DASI or GSI) terminals can print boxed tables directly.

For the convenience of users employing line printers without adequate driving tables or post-filters, there is a special −*TX* command line option to *tbl* which produces output that does not have fractional line motions in it. The only other command line options recognized by *tbl* are −*ms* and −*mm* which are turned into commands to fetch the corresponding macro files; usually it is more convenient to place these arguments on the *troff* part of the command line, but they are accepted by *tbl* as well.

Note that when *eqn* and *tbl* are used together on the same file *tbl* should be used first. If there are no equations within tables, either order works, but it is usually faster to run *tbl* first, since *eqn* normally produces a larger expansion of the input than *tbl*. However, if there are equations within tables (using the *delim* mechanism in *eqn*), *tbl* must be first or the output will be scrambled. Users must also beware of using equations in **n**-style columns; this is nearly

always wrong, since *tbl* attempts to split numerical format items into two parts and this is not possible with equations. The user can defend against this by giving the *delim(xx)* table option; this prevents splitting of numerical columns within the delimiters. For example, if the *eqn* delimiters are *$$*, giving *delim($$)* a numerical column such as "1245 $+- 16$" will be divided after 1245, not after 16.

Tbl limits tables to twenty columns; however, use of more than 16 numerical columns may fail because of limits in *troff*, producing the 'too many number registers' message. *Troff* number registers used by *tbl* must be avoided by the user within tables; these include two-digit names from 31 to 99, and names of the forms #*x*, *x*+, *x*|, ^*x*, and *x*–, where *x* is any lower case letter. The names ##, #–, and #^ are also used in certain circumstances. To conserve number register names, the **n** and **a** formats share a register; hence the restriction above that they may not be used in the same column.

For aid in writing layout macros, *tbl* defines a number register TW which is the table width; it is defined by the time that the ".TE" macro is invoked and may be used in the expansion of that macro. More importantly, to assist in laying out multi-page boxed tables the macro T# is defined to produce the bottom lines and side lines of a boxed table, and then invoked at its end. By use of this macro in the page footer a multi-page table can be boxed. In particular, the *ms* macros can be used to print a multi-page boxed table with a repeated heading by giving the argument H to the ".TS" macro. If the table start macro is written

```
.TS H
```

a line of the form

```
.TH
```

must be given in the table after any table heading (or at the start if none). Material up to the ".TH" is placed at the top of each page of table; the remaining lines in the table are placed on several pages as required. Note that this is *not* a feature of *tbl*, but of the *ms* layout macros.

Examples.

Here are some examples illustrating features of *tbl*. The symbol ⓣ in the input represents a tab character.

Input:

```
.TS
box;
c c c
l l l.
LanguageⓣAuthorsⓣRuns on

FortranⓣManyⓣAlmost anything
PL/1ⓣIBMⓣ360/370
CⓣBTLⓣ11/45,H6000,370
BLISSⓣCarnegie-MellonⓣPDP-10,11
IDSⓣHoneywellⓣH6000
PascalⓣStanfordⓣ370
.TE
```

Output:

Language	Authors	Runs on
Fortran	Many	Almost anything
PL/1	IBM	360/370
C	BTL	11/45,H6000,370
BLISS	Carnegie-Mellon	PDP-10,11
IDS	Honeywell	H6000
Pascal	Stanford	370

Input:

```
.TS
allbox;
c s s
c c c
n n n.
AT&T Common Stock
YearⓉPriceⓉDividend
1971Ⓣ41-54Ⓣ$2.60
2Ⓣ41-54Ⓣ2.70
3Ⓣ46-55Ⓣ2.87
4Ⓣ40-53Ⓣ3.24
5Ⓣ45-52Ⓣ3.40
6Ⓣ51-59Ⓣ.95*
.TE
* (first quarter only)
```

Output:

AT&T Common Stock		
Year	Price	Dividend
1971	41-54	$2.60
2	41-54	2.70
3	46-55	2.87
4	40-53	3.24
5	45-52	3.40
6	51-59	.95*

* (first quarter only)

Input:

```
.TS
box;
c s s
c|c|c
l|l|n.
Major New York Bridges
=
BridgeⓉDesignerⓉLength
_
BrooklynⓉJ. A. RoeblingⓉ1595
ManhattanⓉG. LindenthalⓉ1470
WilliamsburgⓉL. L. BuckⓉ1600
_
QueensboroughⓉPalmer &Ⓣ1182
Ⓣ Hornbostel
_
ⓉⓉ1380
TriboroughⓉO. H. AmmannⓉ_
ⓉⓉ383
_
Bronx WhitestoneⓉO. H. AmmannⓉ2300
Throgs NeckⓉO. H. AmmannⓉ1800
_
George WashingtonⓉO. H. AmmannⓉ3500
.TE
```

Output:

Major New York Bridges		
Bridge	Designer	Length
Brooklyn	J. A. Roebling	1595
Manhattan	G. Lindenthal	1470
Williamsburg	L. L. Buck	1600
Queensborough	Palmer & Hornbostel	1182
		1380
Triborough	O. H. Ammann	
		383
Bronx Whitestone	O. H. Ammann	2300
Throgs Neck	O. H. Ammann	1800
George Washington	O. H. Ammann	3500

Input:

```
.TS
c c
np-2 | n | .
ⓉStack
Ⓣ_
1Ⓣ46
Ⓣ_
2Ⓣ23
Ⓣ_
3Ⓣ15
Ⓣ_
4Ⓣ6.5
Ⓣ_
5Ⓣ2.1
Ⓣ_
.TE
```

Output:

	Stack
1	46
2	23
3	15
4	6.5
5	2.1

Input:

```
.TS
box;
L L L
L L _
L L | LB
L L _
L L L.
januaryⓉfebruaryⓉmarch
aprilⓉmay
juneⓉjulyⓉMonths
augustⓉseptember
octoberⓉnovemberⓉdecember
.TE
```

Output:

january	february	march
april	may	
june	july	**Months**
august	september	
october	november	december

Input:

```
.TS
box;
cfB s s s.
Composition of Foods
_
.T&
c |c s s
c |c s s
c  |c |c |c.
FoodⓉPercent by Weight
\^Ⓣ_
\^ⓉProteinⓉFatⓉCarbo-
\^Ⓣ\^Ⓣ\^Ⓣhydrate
_
.T&
l |n |n |n.
ApplesⓉ.4Ⓣ.5Ⓣ13.0
HalibutⓉ18.4Ⓣ5.2Ⓣ. . .
Lima beansⓉ7.5Ⓣ.8Ⓣ22.0
MilkⓉ3.3Ⓣ4.0Ⓣ5.0
MushroomsⓉ3.5Ⓣ.4Ⓣ6.0
Rye breadⓉ9.0Ⓣ.6Ⓣ52.7
.TE
```

Output:

Composition of Foods			
Food	Percent by Weight		
	Protein	Fat	Carbo-hydrate
Apples	.4	.5	13.0
Halibut	18.4	5.2	...
Lima beans	7.5	.8	22.0
Milk	3.3	4.0	5.0
Mushrooms	3.5	.4	6.0
Rye bread	9.0	.6	52.7

Input:

```
.TS
allbox;
cfI  s   s
c   cw(1i)   cw(1i)
lp9 lp9 lp9.
New York Area Rocks
EraⓉFormationⓉAge (years)
PrecambrianⓉReading ProngⓉ>1 billion
PaleozoicⓉManhattan ProngⓉ400 million
MesozoicⓉT{
.na
Newark Basin, incl.
Stockton, Lockatong, and Brunswick
formations; also Watchungs
and Palisades.
T}Ⓣ200 million
CenozoicⓉCoastal PlainⓉT{
On Long Island 30,000 years;
Cretaceous sediments redeposited
by recent glaciation.
.ad
T}
.TE
```

Output:

New York Area Rocks		
Era	Formation	Age (years)
Precambrian	Reading Prong	>1 billion
Paleozoic	Manhattan Prong	400 million
Mesozoic	Newark Basin, incl. Stockton, Lockatong, and Brunswick formations; also Watchungs and Palisades.	200 million
Cenozoic	Coastal Plain	On Long Island 30,000 years; Cretaceous sediments redeposited by recent glaciation.

Input:

```
.EQ
delim $$
.EN

. . .

.TS
doublebox;
c c
l l.
Name Ⓣ Definition
.sp
.vs +2p
GammaⓉ$GAMMA (z) = int sub 0 sup inf  t sup {z-1} e sup -t dt$
SineⓉ$sin (x) = 1 over 2i ( e sup ix - e sup -ix )$
ErrorⓉ$ roman erf (z) = 2 over sqrt pi int sub 0 sup z e sup {-t sup 2} dt$
BesselⓉ$ J sub 0 (z) = 1 over pi int sub 0 sup pi cos ( z sin theta ) d theta $
ZetaⓉ$ zeta (s) = sum from k=1 to inf k sup -s ~~( Re~s > 1)$
.vs -2p
.TE
```

Output:

Name	Definition
Gamma	$\Gamma(z)=\int_0^\infty t^{z-1}e^{-t}dt$
Sine	$\sin(x)=\frac{1}{2i}(e^{ix}-e^{-ix})$
Error	$\mathrm{erf}(z)=\frac{2}{\sqrt{\pi}}\int_0^z e^{-t^2}dt$
Bessel	$J_0(z)=\frac{1}{\pi}\int_0^\pi \cos(z\sin\theta)d\theta$
Zeta	$\zeta(s)=\sum_{k=1}^\infty k^{-s}\quad(\mathrm{Re}\ s>1)$

Input:

```
.TS
box, tab(:);
cb s s s s
cp-2 s s s s
c||c|c|c|c
c||c|c|c|c
r2||n2|n2|n2|n.
Readability of Text
Line Width and Leading for 10-Point Type
=
Line:Set:1-Point:2-Point:4-Point
Width:Solid:Leading:Leading:Leading
_
9 Pica:\-9.3:\-6.0:\-5.3:\-7.1
14 Pica:\-4.5:\-0.6:\-0.3:\-1.7
19 Pica:\-5.0:\-5.1: 0.0:\-2.0
31 Pica:\-3.7:\-3.8:\-2.4:\-3.6
43 Pica:\-9.1:\-9.0:\-5.9:\-8.8
.TE
```

Output:

Readability of Text Line Width and Leading for 10-Point Type				
Line Width	Set Solid	1-Point Leading	2-Point Leading	4-Point Leading
9 Pica	−9.3	−6.0	−5.3	−7.1
14 Pica	−4.5	−0.6	−0.3	−1.7
19 Pica	−5.0	−5.1	0.0	−2.0
31 Pica	−3.7	−3.8	−2.4	−3.6
43 Pica	−9.1	−9.0	−5.9	−8.8

Input:

```
.TS
c s
cip-2 s
l n
a n.
Some London Transport Statistics
(Year 1964)
Railway route milesⓉ244
TubeⓉ66
Sub-surfaceⓉ22
SurfaceⓉ156
.sp .5
.T&
l r
a r.
Passenger traffic \- railway
JourneysⓉ674 million
Average lengthⓉ4.55 miles
Passenger milesⓉ3,066 million
.T&
l r
a r.
Passenger traffic \- road
JourneysⓉ2,252 million
Average lengthⓉ2.26 miles
Passenger milesⓉ5,094 million
.T&
l n
a n.
.sp .5
VehiclesⓉ12,521
Railway motor carsⓉ2,905
Railway trailer carsⓉ1,269
Total railwayⓉ4,174
OmnibusesⓉ8,347
.T&
l n
a n.
.sp .5
StaffⓉ73,739
Administrative, etc.Ⓣ5,582
Civil engineeringⓉ5,134
Electrical eng.Ⓣ1,714
Mech. eng. \- railwayⓉ4,310
Mech. eng. \- roadⓉ9,152
Railway operationsⓉ8,930
Road operationsⓉ35,946
OtherⓉ2,971
.TE
```

Output:

Some London Transport Statistics	
(Year 1964)	
Railway route miles	244
Tube	66
Sub-surface	22
Surface	156
Passenger traffic – railway	
Journeys	674 million
Average length	4.55 miles
Passenger miles	3,066 million
Passenger traffic – road	
Journeys	2,252 million
Average length	2.26 miles
Passenger miles	5,094 million
Vehicles	12,521
Railway motor cars	2,905
Railway trailer cars	1,269
Total railway	4,174
Omnibuses	8,347
Staff	73,739
Administrative, etc.	5,582
Civil engineering	5,134
Electrical eng.	1,714
Mech. eng. – railway	4,310
Mech. eng. – road	9,152
Railway operations	8,930
Road operations	35,946
Other	2,971

Input:

```
.ps 8
.vs 10p
.TS
center box;
c s s
ci s s
c c c
lB l n.
New Jersey Representatives
(Democrats)
.sp .5
NameⓉOffice addressⓉPhone
.sp .5
James J. FlorioⓉ23 S. White Horse Pike, Somerdale 08083Ⓣ609-627-8222
William J. HughesⓉ2920 Atlantic Ave., Atlantic City 08401Ⓣ609-345-4844
James J. HowardⓉ801 Bangs Ave., Asbury Park 07712Ⓣ201-774-1600
Frank Thompson, Jr.Ⓣ10 Rutgers Pl., Trenton 08618Ⓣ609-599-1619
Andrew MaguireⓉ115 W. Passaic St., Rochelle Park 07662Ⓣ201-843-0240
Robert A. RoeⓉU.S.P.O., 194 Ward St., Paterson 07510Ⓣ201-523-5152
Henry HelstoskiⓉ666 Paterson Ave., East Rutherford 07073Ⓣ201-939-9090
Peter W. Rodino, Jr.ⓉSuite 1435A, 970 Broad St., Newark 07102Ⓣ201-645-3213
Joseph G. MinishⓉ308 Main St., Orange 07050Ⓣ201-645-6363
Helen S. MeynerⓉ32 Bridge St., Lambertville 08530Ⓣ609-397-1830
Dominick V. DanielsⓉ895 Bergen Ave., Jersey City 07306Ⓣ201-659-7700
Edward J. PattenⓉNatl. Bank Bldg., Perth Amboy 08861Ⓣ201-826-4610
.sp .5
.T&
ci s s
lB l n.
(Republicans)
.sp .5v
Millicent FenwickⓉ41 N. Bridge St., Somerville 08876Ⓣ201-722-8200
Edwin B. ForsytheⓉ301 Mill St., Moorestown 08057Ⓣ609-235-6622
Matthew J. RinaldoⓉ1961 Morris Ave., Union 07083Ⓣ201-687-4235
.TE
.ps 10
.vs 12p
```

Output:

New Jersey Representatives *(Democrats)*		
Name	Office address	Phone
James J. Florio	23 S. White Horse Pike, Somerdale 08083	609-627-8222
William J. Hughes	2920 Atlantic Ave., Atlantic City 08401	609-345-4844
James J. Howard	801 Bangs Ave., Asbury Park 07712	201-774-1600
Frank Thompson, Jr.	10 Rutgers Pl., Trenton 08618	609-599-1619
Andrew Maguire	115 W. Passaic St., Rochelle Park 07662	201-843-0240
Robert A. Roe	U.S.P.O., 194 Ward St., Paterson 07510	201-523-5152
Henry Helstoski	666 Paterson Ave., East Rutherford 07073	201-939-9090
Peter W. Rodino, Jr.	Suite 1435A, 970 Broad St., Newark 07102	201-645-3213
Joseph G. Minish	308 Main St., Orange 07050	201-645-6363
Helen S. Meyner	32 Bridge St., Lambertville 08530	609-397-1830
Dominick V. Daniels	895 Bergen Ave., Jersey City 07306	201-659-7700
Edward J. Patten	Natl. Bank Bldg., Perth Amboy 08861	201-826-4610
	(Republicans)	
Millicent Fenwick	41 N. Bridge St., Somerville 08876	201-722-8200
Edwin B. Forsythe	301 Mill St., Moorestown 08057	609-235-6622
Matthew J. Rinaldo	1961 Morris Ave., Union 07083	201-687-4235

This is a paragraph of normal text placed here only to indicate where the left and right margins are. In this way the reader can judge the appearance of centered tables or expanded tables, and observe how such tables are formatted.

Input:

```
.TS
expand;
c s s s
c c c c
l l n n.
Bell Labs Locations
NameⓉAddressⓉArea CodeⓉPhone
HolmdelⓉHolmdel, N. J. 07733Ⓣ201Ⓣ949-3000
Murray HillⓉMurray Hill, N. J. 07974Ⓣ201Ⓣ582-6377
WhippanyⓉWhippany, N. J. 07981Ⓣ201Ⓣ386-3000
Indian HillⓉNaperville, Illinois 60540Ⓣ312Ⓣ690-2000
.TE
```

Output:

Bell Labs Locations			
Name	Address	Area Code	Phone
Holmdel	Holmdel, N. J. 07733	201	949-3000
Murray Hill	Murray Hill, N. J. 07974	201	582-6377
Whippany	Whippany, N. J. 07981	201	386-3000
Indian Hill	Naperville, Illinois 60540	312	690-2000

Input:

```
.TS
box;
cb  s  s  s
c|c|c  s
ltiw(1i) | ltw(2i) | lp8 | lw(1.6i)p8.
Some Interesting Places
_
NameⓉDescriptionⓉPractical Information
_
T{
American Museum of Natural History
T}ⓉT{
The collections fill 11.5 acres (Michelin) or 25 acres (MTA)
of exhibition halls on four floors. There is a full-sized replica
of a blue whale and the world's largest star sapphire (stolen in 1964).
T}ⓉHoursⓉ10-5, ex. Sun 11-5, Wed. to 9
\^Ⓣ\^ⓉLocationⓉT{
Central Park West & 79th St.
T}
\^Ⓣ\^ⓉAdmissionⓉDonation: $1.00 asked
\^Ⓣ\^ⓉSubwayⓉAA to 81st St.
\^Ⓣ\^ⓉTelephoneⓉ212-873-4225
_
Bronx ZooⓉT{
About a mile long and .6 mile wide, this is the largest zoo in America.
A lion eats 18 pounds
of meat a day while a sea lion eats 15 pounds of fish.
T}ⓉHoursⓉT{
10-4:30 winter, to 5:00 summer
T}
\^Ⓣ\^ⓉLocationⓉT{
185th St. & Southern Blvd, the Bronx.
T}
\^Ⓣ\^ⓉAdmissionⓉ$1.00, but Tu,We,Th free
\^Ⓣ\^ⓉSubwayⓉ2, 5 to East Tremont Ave.
\^Ⓣ\^ⓉTelephoneⓉ212-933-1759
_
Brooklyn MuseumⓉT{
Five floors of galleries contain American and ancient art.
There are American period rooms and architectural ornaments saved
from wreckers, such as a classical figure from Pennsylvania Station.
T}ⓉHoursⓉWed-Sat, 10-5, Sun 12-5
\^Ⓣ\^ⓉLocationⓉT{
Eastern Parkway & Washington Ave., Brooklyn.
T}
\^Ⓣ\^ⓉAdmissionⓉFree
\^Ⓣ\^ⓉSubwayⓉ2,3 to Eastern Parkway.
\^Ⓣ\^ⓉTelephoneⓉ212-638-5000
_
T{
New-York Historical Society
T}ⓉT{
All the original paintings for Audubon's
.I
Birds of America
.R
are here, as are exhibits of American decorative arts, New York history,
Hudson River school paintings, carriages, and glass paperweights.
T}ⓉHoursⓉT{
Tues-Fri & Sun, 1-5; Sat 10-5
T}
\^Ⓣ\^ⓉLocationⓉT{
Central Park West & 77th St.
T}
\^Ⓣ\^ⓉAdmissionⓉFree
\^Ⓣ\^ⓉSubwayⓉAA to 81st St.
\^Ⓣ\^ⓉTelephoneⓉ212-873-3400
.TE
```

Output:

Some Interesting Places			
Name	Description	Practical Information	
American Museum of Natural History	The collections fill 11.5 acres (Michelin) or 25 acres (MTA) of exhibition halls on four floors. There is a full-sized replica of a blue whale and the world's largest star sapphire (stolen in 1964).	Hours Location Admission Subway Telephone	10-5, ex. Sun 11-5, Wed. to 9 Central Park West & 79th St. Donation: $1.00 asked AA to 81st St. 212-873-4225
Bronx Zoo	About a mile long and .6 mile wide, this is the largest zoo in America. A lion eats 18 pounds of meat a day while a sea lion eats 15 pounds of fish.	Hours Location Admission Subway Telephone	10-4:30 winter, to 5:00 summer 185th St. & Southern Blvd, the Bronx. $1.00, but Tu,We,Th free 2, 5 to East Tremont Ave. 212-933-1759
Brooklyn Museum	Five floors of galleries contain American and ancient art. There are American period rooms and architectural ornaments saved from wreckers, such as a classical figure from Pennsylvania Station.	Hours Location Admission Subway Telephone	Wed-Sat, 10-5, Sun 12-5 Eastern Parkway & Washington Ave., Brooklyn. Free 2,3 to Eastern Parkway. 212-638-5000
New-York Historical Society	All the original paintings for Audubon's *Birds of America* are here, as are exhibits of American decorative arts, New York history, Hudson River school paintings, carriages, and glass paperweights.	Hours Location Admission Subway Telephone	Tues-Fri & Sun, 1-5; Sat 10-5 Central Park West & 77th St. Free AA to 81st St. 212-873-3400

Acknowledgments.

Many thanks are due to J. C. Blinn, who has done a large amount of testing and assisted with the design of the program. He has also written many of the more intelligible sentences in this document and helped edit all of it. All phototypesetting programs on UNIX are dependent on the work of the late J. F. Ossanna, whose assistance with this program in particular had been most helpful. This program is patterned on a table formatter originally written by J. F. Gimpel. The assistance of T. A. Dolotta, B. W. Kernighan, and J. N. Sturman is gratefully acknowledged.

References.

[1] J. F. Ossanna, *NROFF/TROFF User's Manual*, Computing Science Technical Report No. 54, Bell Laboratories, 1976.

[2] K. Thompson and D. M. Ritchie, "The UNIX Time-Sharing System," *Comm. ACM*, **17**, pp. 365–75 (1974).

[3] B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. ACM*, **18**, pp. 151–57 (1975).

[4] M. E. Lesk, *Typing Documents on UNIX*, UNIX Programmer's Manual, Volume 2.

[5] M. E. Lesk and B. W. Kernighan, *Computer Typesetting of Technical Journals on UNIX, Proc. AFIPS NCC*, vol. 46, pp. 879-888 (1977).

[6] J. R. Mashey and D. W. Smith, "Documentation Tools and Techniques," *Proc. 2nd Int. Conf. on Software Engineering*, pp. 177-181 (October, 1976).

List of Tbl Command Characters and Words

Command	*Meaning*	*Section*
a A	Alphabetic subcolumn	2
allbox	Draw box around all items	1
b B	Boldface item	2
box	Draw box around table	1
c C	Centered column	2
center	Center table in page	1
doublebox	Doubled box around table	1
e E	Equal width columns	2
expand	Make table full line width	1
f F	Font change	2
i I	Italic item	2
l L	Left adjusted column	2
n N	Numerical column	2
nnn	Column separation	2
p P	Point size change	2
r R	Right adjusted column	2
s S	Spanned item	2
t T	Vertical spanning at top	2
tab (*x*)	Change data separator character	1
T{ T}	Text block	3
v V	Vertical spacing change	2
w W	Minimum width value	2
.*xx*	Included *troff* command	3
|	Vertical line	2
||	Double vertical line	2
^	Vertical span	2
\^	Vertical span	3
=	Double horizontal line	2,3
_	Horizontal line	2,3
_	Short horizontal line	3
\R*x*	Repeat character	3

Typesetting Mathematics — User's Guide (Second Edition)

Brian W. Kernighan and Lorinda L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This is the user's guide for a system for typesetting mathematics, using the phototypesetters on the UNIX† and GCOS operating systems.

Mathematical expressions are described in a language designed to be easy to use by people who know neither mathematics nor typesetting. Enough of the language to set in-line expressions like $\lim_{x \to \pi/2} (\tan x)^{\sin 2x} = 1$ or display equations like

$$G(z) = e^{\ln G(z)} = \exp\left(\sum_{k \geq 1} \frac{S_k z^k}{k} \right) = \prod_{k \geq 1} e^{S_k z^k / k}$$

$$= \left(1 + S_1 z + \frac{S_1^2 z^2}{2!} + \cdots \right) \left(1 + \frac{S_2 z^2}{2} + \frac{S_2^2 z^4}{2^2 \cdot 2!} + \cdots \right) \cdots$$

$$= \sum_{m \geq 0} \left(\sum_{\substack{k_1, k_2, \ldots, k_m \geq 0 \\ k_1 + 2k_2 + \cdots + mk_m = m}} \frac{S_1^{k_1}}{1^{k_1} k_1 !} \, \frac{S_2^{k_2}}{2^{k_2} k_2 !} \, \cdots \, \frac{S_m^{k_m}}{m^{k_m} k_m !} \right) z^m$$

can be learned in an hour or so.

The language interfaces directly with the phototypesetting language TROFF, so mathematical expressions can be embedded in the running text of a manuscript, and the entire document produced in one process. This user's guide is an example of its output.

The same language may be used with the UNIX formatter NROFF to set mathematical expressions on DASI and GSI terminals and Model 37 teletypes.

August 15, 1978

†UNIX is a Trademark of Bell Laboratories.

Typesetting Mathematics — User's Guide (Second Edition)

Brian W. Kernighan and Lorinda L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

EQN is a program for typesetting mathematics on the Graphics Systems phototypesetters on UNIX and GCOS. The EQN language was designed to be easy to use by people who know neither mathematics nor typesetting. Thus EQN knows relatively little about mathematics. In particular, mathematical symbols like +, −, ×, parentheses, and so on have no special meanings. EQN is quite happy to set garbage (but it will look good).

EQN works as a preprocessor for the typesetter formatter, TROFF[1], so the normal mode of operation is to prepare a document with both mathematics and ordinary text interspersed, and let EQN set the mathematics while TROFF does the body of the text.

On UNIX, EQN will also produce mathematics on DASI and GSI terminals and on Model 37 teletypes. The input is identical, but you have to use the programs NEQN and NROFF instead of EQN and TROFF. Of course, some things won't look as good because terminals don't provide the variety of characters, sizes and fonts that a typesetter does, but the output is usually adequate for proofreading.

To use EQN on UNIX,

```
eqn files | troff
```

GCOS use is discussed in section 26.

2. Displayed Equations

To tell EQN where a mathematical expression begins and ends, we mark it with lines beginning .EQ and .EN. Thus if you type the lines

```
.EQ
x=y+z
.EN
```

your output will look like

$$x=y+z$$

The .EQ and .EN are copied through untouched; they are not otherwise processed by EQN. This means that you have to take care of things like centering, numbering, and so on yourself. The most common way is to use the TROFF and NROFF macro package package '−ms' developed by M. E. Lesk[3], which allows you to center, indent, left-justify and number equations.

With the '−ms' package, equations are centered by default. To left-justify an equation, use .EQ L instead of .EQ. To indent it, use .EQ I. Any of these can be followed by an arbitrary 'equation number' which will be placed at the right margin. For example, the input

```
.EQ I (3.1a)
x = f(y/2) + y/2
.EN
```

produces the output

$$x=f(y/2)+y/2 \qquad (3.1a)$$

There is also a shorthand notation so in-line expressions like π_i^2 can be entered without .EQ and .EN. We will talk about it in section 19.

3. Input spaces

Spaces and newlines within an expression are thrown away by EQN. (Normal text is left absolutely alone.) Thus between .EQ and .EN,

```
x=y+z
```

and

```
x = y + z
```

and

```
x  =  y
  + z
```

and so on all produce the same output

$$x=y+z$$

You should use spaces and newlines freely to make your input equations readable and easy to edit. In particular, very long lines are a bad idea, since they are often hard to fix if you make a mistake.

4. Output spaces

To force extra spaces into the *output*, use a tilde "~" for each space you want:

```
x~=~y~+~z
```

gives

$$x \ = \ y \ + \ z$$

You can also use a circumflex "^", which gives a space half the width of a tilde. It is mainly useful for fine-tuning. Tabs may also be used to position pieces of an expression, but the tab stops must be set by TROFF commands.

5. Symbols, Special Names, Greek

EQN knows some mathematical symbols, some mathematical names, and the Greek alphabet. For example,

```
x=2 pi int sin ( omega t)dt
```

produces

$$x=2\pi\int\sin(\omega t)\,dt$$

Here the spaces in the input are **necessary** to tell EQN that *int, pi, sin* and *omega* are separate entities that should get special treatment. The *sin*, digit 2, and parentheses are set in roman type instead of italic; *pi* and *omega* are made Greek; and *int* becomes the integral sign.

When in doubt, leave spaces around separate parts of the input. A *very* common error is to type *f(pi)* without leaving spaces on both sides of the *pi*. As a result, EQN does not recognize *pi* as a special word, and it appears as $f(pi)$ instead of $f(\pi)$.

A complete list of EQN names appears in section 23. Knowledgeable users can also use TROFF four-character names for anything EQN doesn't know about, like \(*bs* for the Bell System sign Ⓑ.

6. Spaces, Again

The only way EQN can deduce that some sequence of letters might be special is if that sequence is separated from the letters on either side of it. This can be done by surrounding a special word by ordinary spaces (or tabs or newlines), as we did in the previous section.

You can also make special words stand out by surrounding them with tildes or circumflexes:

```
x~=~2~pi~int~sin~(~omega~t~)~dt
```

is much the same as the last example, except that the tildes not only separate the magic words like *sin, omega*, and so on, but also add extra spaces, one space per tilde:

$$x \ = \ 2 \ \pi \ \int \ \sin \ (\ \omega \ t \ ) \ dt$$

Special words can also be separated by braces { } and double quotes "...", which have special meanings that we will see soon.

7. Subscripts and Superscripts

Subscripts and superscripts are obtained with the words *sub* and *sup*.

```
x sup 2 + y sub k
```

gives

$$x^2+y_k$$

EQN takes care of all the size changes and vertical motions needed to make the output look right. The words *sub* and *sup* must be surrounded by spaces; *x sub2* will give you *xsub2* instead of x_2. Furthermore, don't forget to leave a space (or a tilde, etc.) to mark the end of a subscript or superscript. A common error is to say something like

```
y = (x sup 2)+1
```

which causes

$$y=(x^{2)+1}$$

instead of the intended

$$y=(x^2)+1$$

Subscripted subscripts and superscripted superscripts also work:

x sub i sub 1

is

$$x_{i_1}$$

A subscript and superscript on the same thing are printed one above the other if the subscript comes *first:*

x sub i sup 2

is

$$x_i^2$$

Other than this special case, *sub* and *sup* group to the right, so *x sup y sub z* means x^{y_z}, not ${x^y}_z$.

8. Braces for Grouping

Normally, the end of a subscript or superscript is marked simply by a blank (or tab or tilde, etc.) What if the subscript or superscript is something that has to be typed with blanks in it? In that case, you can use the braces { and } to mark the beginning and end of the subscript or superscript:

e sup {i omega t}

is

$$e^{i \omega t}$$

Rule: Braces can *always* be used to force EQN to treat something as a unit, or just to make your intent perfectly clear. Thus:

x sub {i sub 1} sup 2

is

$$x_{i_1}^2$$

with braces, but

x sub i sub 1 sup 2

is

$$x_{i_1^2}$$

which is rather different.

Braces can occur within braces if necessary:

e sup {i pi sup {rho +1}}

is

The general rule is that anywhere you could use some single thing like *x*, you can use an arbitrarily complicated thing if you enclose it in braces. EQN will look after all the details of positioning it and making it the right size.

In all cases, make sure you have the right number of braces. Leaving one out or adding an extra will cause EQN to complain bitterly.

Occasionally you will have to print braces. To do this, enclose them in double quotes, like "{". Quoting is discussed in more detail in section 14.

9. Fractions

To make a fraction, use the word *over:*

a+b over 2c =1

gives

$$\frac{a+b}{2c}=1$$

The line is made the right length and positioned automatically. Braces can be used to make clear what goes over what:

{alpha + beta} over {sin (x)}

is

$$\frac{\alpha+\beta}{\sin(x)}$$

What happens when there is both an *over* and a *sup* in the same expression? In such an apparently ambiguous case, EQN does the *sup* before the *over*, so

−b sup 2 over pi

is $\frac{-b^2}{\pi}$ instead of $-b^{\frac{2}{\pi}}$ The rules which decide which operation is done first in cases like this are summarized in section 23. When in doubt, however, *use braces* to make clear what goes with what.

10. Square Roots

To draw a square root, use *sqrt:*

sqrt a+b + 1 over sqrt {ax sup 2 +bx+c}

is

$$\sqrt{a+b}+\frac{1}{\sqrt{ax^2+bx+c}}$$

Warning — square roots of tall quantities look lousy, because a root-sign big enough to cover the quantity is too dark and heavy:

sqrt {a sup 2 over b sub 2}

is

$$\sqrt{\frac{a^2}{b_2}}$$

Big square roots are generally better written as something to the power ½:

$$(a^2/b_2)^{1/2}$$

which is

(a sup 2 /b sub 2) sup half

11. Summation, Integral, Etc.

Summations, integrals, and similar constructions are easy:

sum from i=0 to {i= inf} x sup i

produces

$$\sum_{i=0}^{i=\infty} x^i$$

Notice that we used braces to indicate where the upper part $i=\infty$ begins and ends. No braces were necessary for the lower part $i=0$, because it contained no blanks. The braces will never hurt, and if the *from* and *to* parts contain any blanks, you must use braces around them.

The *from* and *to* parts are both optional, but if both are used, they have to occur in that order.

Other useful characters can replace the *sum* in our example:

int prod union inter

become, respectively,

$$\int \quad \prod \quad \bigcup \quad \bigcap$$

Since the thing before the *from* can be anything, even something in braces, *from-to* can often be used in unexpected ways:

lim from {n -> inf} x sub n =0

is

$$\lim_{n \to \infty} x_n = 0$$

12. Size and Font Changes

By default, equations are set in 10-point type (the same size as this guide), with standard mathematical conventions to determine what characters are in roman and what in italic. Although EQN makes a valiant attempt to use esthetically pleasing sizes and fonts, it is not perfect. To change sizes and fonts, use *size n* and *roman, italic, bold* and *fat.* Like *sub* and *sup,* size and font changes affect only the thing that follows them, and revert to the normal situation at the end of it. Thus

bold x y

is

$$\mathbf{x}y$$

and

size 14 bold x = y +
  size 14 {alpha + beta}

gives

$$\mathbf{x}=y+\alpha+\beta$$

As always, you can use braces if you want to affect something more complicated than a single letter. For example, you can change the size of an entire equation by

size 12 { ... }

Legal sizes which may follow *size* are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, 36. You can also change the size *by* a given amount; for example, you can say *size +2* to make the size two points bigger, or *size −3* to make it three points smaller. This has the advantage that you don't have to know what the current size is.

If you are using fonts other than roman, italic and bold, you can say *font X* where *X* is a one character TROFF name or number for the font. Since EQN is tuned for roman, italic and bold, other fonts may not give quite as good an appearance.

The *fat* operation takes the current font and widens it by overstriking: *fat grad* is ∇ and *fat {x sub i}* is x_i.

If an entire document is to be in a non-standard size or font, it is a severe nuisance to have to write out a size and font change for each equation. Accordingly, you can set a "global" size or font which

thereafter affects all equations. At the beginning of any equation, you might say, for instance,

```
.EQ
gsize 16
gfont R
...
.EN
```

to set the size to 16 and the font to roman thereafter. In place of R, you can use any of the TROFF font names. The size after *gsize* can be a relative change with + or −.

Generally, *gsize* and *gfont* will appear at the beginning of a document but they can also appear thoughout a document: the global font and size can be changed as often as needed. For example, in a footnote‡ you will typically want the size of equations to match the size of the footnote text, which is two points smaller than the main text. Don't forget to reset the global size at the end of the footnote.

13. Diacritical Marks

To get funny marks on top of letters, there are several words:

x dot	$\dot{x}$
x dotdot	$\ddot{x}$
x hat	$\hat{x}$
x tilde	$\tilde{x}$
x vec	$\vec{x}$
x dyad	$\overleftrightarrow{x}$
x bar	$\bar{x}$
x under	$\underline{x}$

The diacritical mark is placed at the right height. The *bar* and *under* are made the right length for the entire construct, as in $\overline{x+y+z}$; other marks are centered.

14. Quoted Text

Any input entirely within quotes ("...") is not subject to any of the font changes and spacing adjustments normally done by the equation setter. This provides a way to do your own spacing and adjusting if needed:

```
italic "sin(x)" + sin (x)
```

is

sin(x)+sin(*x*)

Quotes are also used to get braces and other EQN keywords printed:

```
"{ size alpha }"
```

is

{ *size alpha* }

and

```
roman "{ size alpha }"
```

is

{ size alpha }

The construction "" is often used as a place-holder when grammatically EQN needs something, but you don't actually want anything in your output. For example, to make ${}^2\mathrm{He}$, you can't just type *sup 2 roman He* because a *sup* has to be a superscript *on* something. Thus you must say

```
"" sup 2 roman He
```

To get a literal quote use "\"". TROFF characters like \(*bs* can appear unquoted, but more complicated things like horizontal and vertical motions with *h* and *v* should always be quoted. (If you've never heard of *h* and *v*, ignore this section.)

15. Lining Up Equations

Sometimes it's necessary to line up a series of equations at some horizontal position, often at an equals sign. This is done with two operations called *mark* and *lineup*.

The word *mark* may appear once at any place in an equation. It remembers the horizontal position where it appeared. Successive equations can contain one occurrence of the word *lineup*. The place where *lineup* appears is made to line up with the place marked by the previous *mark* if at all possible. Thus, for example, you can say

‡Like this one, in which we have a few random expressions like x_i and π^2. The sizes for these were set by the command *gsize* −2.

```
.EQ I
x+y mark = z
.EN
.EQ I
x lineup = 1
.EN
```

to produce

$x+y=z$

$\quad x=1$

For reasons too complicated to talk about, when you use EQN and '−ms', use either .EQ I or .EQ L. mark and *lineup* don't work with centered equations. Also bear in mind that *mark* doesn't look ahead;

```
x mark =1
...
x+y lineup =z
```

isn't going to work, because there isn't room for the $x+y$ part after the *mark* remembers where the x is.

16. Big Brackets, Etc.

To get big brackets [ ], braces { }, parentheses ( ), and bars || around things, use the *left* and *right* commands:

```
left { a over b + 1 right }
~=~ left ( c over d right )
+ left [ e right ]
```

is

$$\left\{ \frac{a}{b}+1 \right\} = \left(\frac{c}{d} \right) + \left[e \right]$$

The resulting brackets are made big enough to cover whatever they enclose. Other characters can be used besides these, but the are not likely to look very good. One exception is the *floor* and *ceiling* characters:

```
left floor x over y right floor
<= left ceiling a over b right ceiling
```

produces

$$\left\lfloor \frac{x}{y} \right\rfloor \leqslant \left\lceil \frac{a}{b} \right\rceil$$

Several warnings about brackets are in order. First, braces are typically bigger than brackets and parentheses, because they are made up of three, five, seven, etc., pieces, while brackets can be made up of two, three, etc. Second, big left and right parentheses often look poor, because the character set is poorly designed.

The *right* part may be omitted: a "left something" need not have a corresponding "right something". If the *right* part is omitted, put braces around the thing you want the left bracket to encompass. Otherwise, the resulting brackets may be too large.

If you want to omit the *left* part, things are more complicated, because technically you can't have a *right* without a corresponding *left*. Instead you have to say

```
left "" ..... right )
```

for example. The *left* "" means a "left nothing". This satisfies the rules without hurting your output.

17. Piles

There is a general facility for making vertical piles of things; it comes in several flavors. For example:

```
A ~=~ left [
 pile { a above b above c }
 ~~ pile { x above y above z }
right ]
```

will make

$$A = \begin{bmatrix} a & x \\ b & y \\ c & z \end{bmatrix}$$

The elements of the pile (there can be as many as you want) are centered one above another, at the right height for most purposes. The keyword *above* is used to separate the pieces; braces are used around the entire list. The elements of a pile can be as complicated as needed, even containing more piles.

Three other forms of pile exist: *lpile* makes a pile with the elements left-justified; *rpile* makes a right-justified pile; and *cpile* makes a centered pile, just like *pile*. The vertical spacing between the pieces is somewhat larger for *l*-, *r*- and *cpiles* than it is for ordinary piles.

```
roman sign (x)~=~
left {
  lpile {1 above 0 above −1}
  ~~ lpile
  {if~x>0 above if~x=0 above if~x<0}
```

makes

$$\mathrm{sign}(x) = \begin{cases} 1 & \text{if } x>0 \\ 0 & \text{if } x=0 \\ -1 & \text{if } x<0 \end{cases}$$

Notice the left brace without a matching right one.

18. Matrices

It is also possible to make matrices. For example, to make a neat array like

$$\begin{matrix} x_i & x^2 \\ y_i & y^2 \end{matrix}$$

you have to type

```
matrix {
  ccol { x sub i above y sub i }
  ccol { x sup 2 above y sup 2 }
}
```

This produces a matrix with two centered columns. The elements of the columns are then listed just as for a pile, each element separated by the word *above.* You can also use *lcol* or *rcol* to left or right adjust columns. Each column can be separately adjusted, and there can be as many columns as you like.

The reason for using a matrix instead of two adjacent piles, by the way, is that if the elements of the piles don't all have the same height, they won't line up properly. A matrix forces them to line up, because it looks at the entire structure before deciding what spacing to use.

A word of warning about matrices — *each column must have the same number of elements in it.* The world will end if you get this wrong.

19. Shorthand for In-line Equations

In a mathematical document, it is necessary to follow mathematical conventions not just in display equations, but also in the body of the text, for example by making variable names like x italic. Although this could be done by surrounding the appropriate parts with .EQ and .EN, the continual repetition of .EQ and .EN is a nuisance. Furthermore, with '−ms', .EQ and .EN imply a displayed equation.

EQN provides a shorthand for short in-line expressions. You can define two characters to mark the left and right ends of an in-line equation, and then type expressions right in the middle of text lines. To set both the left and right characters to dollar signs, for example, add to the beginning of your document the three lines

```
.EQ
delim $$
.EN
```

Having done this, you can then say things like

> Let $alpha sub i$ be the primary variable, and let $beta$ be zero. Then we can show that $x sub 1$ is $>=0$.

This works as you might expect — spaces, newlines, and so on are significant in the text, but not in the equation part itself. Multiple equations can occur in a single input line.

Enough room is left before and after a line that contains in-line expressions that something like $\sum_{i=1}^{n} x_i$ does not interfere with the lines surrounding it.

To turn off the delimiters,

```
.EQ
delim off
.EN
```

Warning: don't use braces, tildes, circumflexes, or double quotes as delimiters — chaos will result.

20. Definitions

EQN provides a facility so you can give a frequently-used string of characters a name, and thereafter just type the name instead of the whole string. For example, if the sequence

```
x sub i sub 1 + y sub i sub 1
```

appears repeatedly throughout a paper, you can save re-typing it each time by defining it like this:

```
define xy 'x sub i sub 1 + y sub i sub 1'
```

This makes *xy* a shorthand for whatever characters occur between the single quotes in the definition. You can use any character

instead of quote to mark the ends of the definition, so long as it doesn't appear inside the definition.

Now you can use *xy* like this:

```
.EQ
f(x) = xy ...
.EN
```

and so on. Each occurrence of *xy* will expand into what it was defined as. Be careful to leave spaces or their equivalent around the name when you actually use it, so EQN will be able to identify it as special.

There are several things to watch out for. First, although definitions can use previous definitions, as in

```
.EQ
define xi  ' x sub i '
define xi1  ' xi sub 1 '
.EN
```

don't define something in terms of itself. A favorite error is to say

```
define X  ' roman X '
```

This is a guaranteed disaster, since X *is* now defined in terms of itself. If you say

```
define X  ' roman "X" '
```

however, the quotes protect the second X, and everything works fine.

EQN keywords can be redefined. You can make / mean *over* by saying

```
define /  ' over '
```

or redefine *over* as / with

```
define over  ' / '
```

If you need different things to print on a terminal and on the typesetter, it is sometimes worth defining a symbol differently in NEQN and EQN. This can be done with *ndefine* and *tdefine*. A definition made with *ndefine* only takes effect if you are running NEQN; if you use *tdefine*, the definition only applies for EQN. Names defined with plain *define* apply to both EQN and NEQN.

21. Local Motions

Although EQN tries to get most things at the right place on the paper, it isn't perfect, and occasionally you will need to tune the output to make it just right. Small extra horizontal spaces can be obtained with tilde and circumflex. You can also say *back n* and *fwd n* to move small amounts horizontally. *n* is how far to move in 1/100's of an em (an em is about the width of the letter 'm'.) Thus *back 50* moves back about half the width of an m. Similarly you can move things up or down with *up n* and *down n*. As with *sub* or *sup*, the local motions affect the next thing in the input, and this can be something arbitrarily complicated if it is enclosed in braces.

22. A Large Example

Here is the complete source for the three display equations in the abstract of this guide.

```
.EQ I
G(z)~mark =~ e sup { ln ~ G(z) }
~=~ exp left (
sum from k>=1 {S sub k z sup k} over k right )
~=~  prod from k>=1 e sup {S sub k z sup k /k}
.EN
.EQ I
lineup = left ( 1 + S sub 1 z +
{ S sub 1 sup 2 z sup 2 } over 2! + ... right )
left ( 1+ { S sub 2 z sup 2 } over 2
+ { S sub 2 sup 2 z sup 4 } over { 2 sup 2 cdot 2! }
+ ... right ) ...
.EN
.EQ I
lineup =  sum from m>=0 left (
sum from
pile { k sub 1 ,k sub 2 ...., k sub m  >=0
above
k sub 1 +2k sub 2 + ... +mk sub m =m}
{ S sub 1 sup {k sub 1} } over {1 sup k sub 1 k sub 1 ! } ~
{ S sub 2 sup {k sub 2} } over {2 sup k sub 2 k sub 2 ! } ~
...
{ S sub m sup {k sub m} } over {m sup k sub m k sub m ! }
right ) z sup m
.EN
```

23. Keywords, Precedences, Etc.

If you don't use braces, EQN will do operations in the order shown in this list.

dyad vec under bar tilde hat dot dotdot
fwd back down up
fat roman italic bold size
sub sup sqrt over
from to

These operations group to the left:

over sqrt left right

All others group to the right.

Digits, parentheses, brackets, punctuation marks, and these mathematical words are converted to Roman font when encountered:

> sin cos tan sinh cosh tanh arc
> max min lim log ln exp
> Re Im and if for det

These character sequences are recognized and translated as shown.

>=	$\geq$
<=	$\leq$
==	$\equiv$
!=	$\neq$
+-	$\pm$
->	$\rightarrow$
<-	$\leftarrow$
<<	$<<$
>>	$>>$
inf	∞
partial	∂
half	½
prime	$'$
approx	$\approx$
nothing	
cdot	$\cdot$
times	$\times$
del	∇
grad	∇
...	$\cdots$
,...,	$,\cdots,$
sum	$\sum$
int	$\int$
prod	$\prod$
union	$\bigcup$
inter	$\bigcap$

To obtain Greek letters, simply spell them out in whatever case you want:

DELTA	Δ	iota	ι
GAMMA	Γ	kappa	κ
LAMBDA	Λ	lambda	λ
OMEGA	Ω	mu	μ
PHI	Φ	nu	ν
PI	Π	omega	ω
PSI	Ψ	omicron	o
SIGMA	Σ	phi	ϕ
THETA	Θ	pi	π
UPSILON	Υ	psi	ψ
XI	Ξ	rho	ρ
alpha	α	sigma	σ

beta	β	tau	τ
chi	χ	theta	θ
delta	δ	upsilon	υ
epsilon	ϵ	xi	ξ
eta	η	zeta	ζ
gamma	γ		

These are all the words known to EQN (except for characters with names), together with the section where they are discussed.

above	17, 18	lpile	17
back	21	mark	15
bar	13	matrix	18
bold	12	ndefine	20
ccol	18	over	9
col	18	pile	17
cpile	17	rcol	18
define	20	right	16
delim	19	roman	12
dot	13	rpile	17
dotdot	13	size	12
down	21	sqrt	10
dyad	13	sub	7
fat	12	sup	7
font	12	tdefine	20
from	11	tilde	13
fwd	21	to	11
gfont	12	under	13
gsize	12	up	21
hat	13	vec	13
italic	12	~ , ^	4, 6
lcol	18	{ }	8
left	16	"..."	8, 14
lineup	15		

24. Troubleshooting

If you make a mistake in an equation, like leaving out a brace (very common) or having one too many (very common) or having a *sup* with nothing before it (common), EQN will tell you with the message

> *syntax error between lines x and y, file z*

where x and y are approximately the lines between which the trouble occurred, and z is the name of the file in question. The line numbers are approximate — look nearby as well. There are also self-explanatory messages that arise if you leave out a quote or try to run EQN on a non-existent file.

If you want to check a document before actually printing it (on UNIX only),

eqn files >/dev/null

will throw away the output but print the messages.

If you use something like dollar signs as delimiters, it is easy to leave one out. This causes very strange troubles. The program *checkeq* (on GCOS, use *./checkeq* instead) checks for misplaced or missing dollar signs and similar troubles.

In-line equations can only be so big because of an internal buffer in TROFF. If you get a message "word overflow", you have exceeded this limit. If you print the equation as a displayed equation this message will usually go away. The message "line overflow" indicates you have exceeded an even bigger buffer. The only cure for this is to break the equation into two separate ones.

On a related topic, EQN does not break equations by itself — you must split long equations up across multiple lines by yourself, marking each by a separate .EQ ... .EN sequence. EQN does warn about equations that are too long to fit on one line.

25. Use on UNIX

To print a document that contains mathematics on the UNIX typesetter,

eqn files | troff

If there are any TROFF options, they go after the TROFF part of the command. For example,

eqn files | troff −ms

To run the same document on the GCOS typesetter, use

eqn files | troff −g (other options) | gcat

A compatible version of EQN can be used on devices like teletypes and DASI and GSI terminals which have half-line forward and reverse capabilities. To print equations on a Model 37 teletype, for example, use

neqn files | nroff

The language for equations recognized by NEQN is identical to that of EQN, although of course the output is more restricted.

To use a GSI or DASI terminal as the output device,

neqn files | nroff −T*x*

where *x* is the terminal type you are using, such as *300* or *300S*.

EQN and NEQN can be used with the TBL program[2] for setting tables that contain mathematics. Use TBL before [N]EQN, like this:

tbl files | eqn | troff
tbl files | neqn | nroff

26. Acknowledgments

We are deeply indebted to J. F. Ossanna, the author of TROFF, for his willingness to extend TROFF to make our task easier, and for his continuous assistance during the development and evolution of EQN. We are also grateful to A. V. Aho for advice on language design, to S. C. Johnson for assistance with the YACC compiler-compiler, and to all the EQN users who have made helpful suggestions and criticisms.

References

[1] J. F. Ossanna, "NROFF/TROFF User's Manual", Bell Laboratories Computing Science Technical Report #54, 1976.

[2] M. E. Lesk, "Typing Documents on UNIX", Bell Laboratories, 1976.

[3] M. E. Lesk, "TBL — A Program for Setting Tables", Bell Laboratories Computing Science Technical Report #49, 1976.

Some Applications of Inverted Indexes on the UNIX System

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction.

The UNIX† system has many utilities (e.g. *grep, awk, lex, egrep, fgrep, ...*) to search through files of text, but most of them are based on a linear scan through the entire file, using some deterministic automaton. This memorandum discusses a program which uses inverted indexes[1] and can thus be used on much larger data bases.

As with any indexing system, of course, there are some disadvantages; once an index is made, the files that have been indexed can not be changed without remaking the index. Thus applications are restricted to those making many searches of relatively stable data. Furthermore, these programs depend on hashing, and can only search for exact matches of whole keywords. It is not possible to look for arithmetic or logical expressions (e.g. "date greater than 1970") or for regular expression searching such as that in *lex*.[2]

Currently there are two uses of this software, the *refer* preprocessor to format references, and the *lookall* command to search through all text files on the UNIX system.

The remaining sections of this memorandum discuss the searching programs and their uses. Section 2 explains the operation of the searching algorithm and describes the data collected for use with the *lookall* command. The more important application, *refer* has a user's description in section 3. Section 4 goes into more detail on reference files for the benefit of those who wish to add references to data bases or write new *troff* macros for use with *refer*. The options to make *refer* collect identical citations, or otherwise relocate and adjust references, are described in section 5. The UNIX manual sections for *refer, lookall,* and associated commands are attached as appendices.

2. Searching.

The indexing and searching process is divided into two phases, each made of two parts. These are shown below.

A. Construct the index.

   (1) Find keys — turn the input files into a sequence of tags and keys, where each tag identifies a distinct item in the input and the keys for each such item are the strings under which it is to be indexed.

   (2) Hash and sort — prepare a set of inverted indexes from which, given a set of keys, the appropriate item tags can be found quickly.

B. Retrieve an item in response to a query.

---

†UNIX is a Trademark of Bell Laboratories.

1. D. Knuth, *The Art of Computer Programming: Vol. 3, Sorting and Searching*, Addison-Wesley, Reading, Mass. (1977). See section 6.5.

2. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (D).

(3) Search — Given some keys, look through the files prepared by the hashing and sorting facility and derive the appropriate tags.

(4) Deliver — Given the tags, find the original items. This completes the searching process.

The first phase, making the index, is presumably done relatively infrequently. It should, of course, be done whenever the data being indexed change. In contrast, the second phase, retrieving items, is presumably done often, and must be rapid.

An effort is made to separate code which depends on the data being handled from code which depends on the searching procedure. The search algorithm is involved only in steps (2) and (3), while knowledge of the actual data files is needed only by steps (1) and (4). Thus it is easy to adapt to different data files or different search algorithms.

To start with, it is necessary to have some way of selecting or generating keys from input files. For dealing with files that are basically English, we have a key-making program which automatically selects words and passes them to the hashing and sorting program (step 2). The format used has one line for each input item, arranged as follows:

name:start,length (tab) key1 key2 key3 ...

where *name* is the file name, *start* is the starting byte number, and *length* is the number of bytes in the entry.

These lines are the only input used to make the index. The first field (the file name, byte position, and byte count) is the tag of the item and can be used to retrieve it quickly. Normally, an item is either a whole file or a section of a file delimited by blank lines. After the tab, the second field contains the keys. The keys, if selected by the automatic program, are any alphanumeric strings which are not among the 100 most frequent words in English and which are not entirely numeric (except for four-digit numbers beginning 19, which are accepted as dates). Keys are truncated to six characters and converted to lower case. Some selection is needed if the original items are ver lrge. We normally just take the first *n* keys, with *n* less than 100 or so; this replaces any attempt at intelligent selection. One file in our system is a complete English dictionary; it would presumably be retrieved for all queries.

To generate an inverted index to the list of record tags and keys, the keys are hashed and sorted to produce an index. What is wanted, ideally, is a series of lists showing the tags associated with each key. To condense this, what is actually produced is a list showing the tags associated with each hash code, and thus with some set of keys. To speed up access and further save space, a set of three or possibly four files is produced. These files are:

File	Contents
entry	Pointers to posting file for each hash code
posting	Lists of tag pointers for each hash code
tag	Tags for each item
key	Keys for each item (optional)

The posting file comprises the real data: it contains a sequence of lists of items posted under each hash code. To speed up searching, the entry file is an array of pointers into the posting file, one per potential hash code. Furthermore, the items in the lists in the posting file are not referred to by their complete tag, but just by an address in the tag file, which gives the complete tags. The key file is optional and contains a copy of the keys used in the indexing.

The searching process starts with a query, containing several keys. The goal is to obtain all items which were indexed under these keys. The query keys are hashed, and the pointers in the entry file used to access the lists in the posting file. These lists are addresses in the tag file of documents posted under the hash codes derived from the query. The common items from

all lists are determined; this must include the items indexed by every key, but may also contain some items which are false drops, since items referenced by the correct hash codes need not actually have contained the correct keys. Normally, if there are several keys in the query, there are not likely to be many false drops in the final combined list even though each hash code is somewhat ambiguous. The actual tags are then obtained from the tag file, and to guard against the possibility that an item has false-dropped on some hash code in the query, the original items are normally obtained from the delivery program (4) and the query keys checked against them by string comparison.

Usually, therefore, the check for bad drops is made against the original file. However, if the key derivation procedure is complex, it may be preferable to check against the keys fed to program (2). In this case the optional key file which contains the keys associated with each item is generated, and the item tag is supplemented by a string

;start,length

which indicates the starting byte number in the key file and the length of the string of keys for each item. This file is not usually necessary with the present key-selection program, since the keys always appear in the original document.

There is also an option (-C*n*) for coordination level searching. This retrieves items which match all but *n* of the query keys. The items are retrieved in the order of the number of keys that they match. Of course, *n* must be less than the number of query keys (nothing is retrieved unless it matches at least one key).

As an example, consider one set of 4377 references, comprising 660,000 bytes. This included 51,000 keys, of which 5,900 were distinct keys. The hash table is kept full to save space (at the expense of time); 995 of 997 possible hash codes were used. The total set of index files (no key file) included 171,000 bytes, about 26% of the original file size. It took 8 minutes of processor time to hash, sort, and write the index. To search for a single query with the resulting index took 1.9 seconds of processor time, while to find the same paper with a sequential linear search using *grep* (reading all of the tags and keys) took 12.3 seconds of processor time.

We have also used this software to index all of the English stored on our UNIX system. This is the index searched by the *lookall* command. On a typical day there were 29,000 files in our user file system, containing about 152,000,000 bytes. Of these 5,300 files, containing 32,000,000 bytes (about 21%) were English text. The total number of 'words' (determined mechanically) was 5,100,000. Of these 227,000 were selected as keys; 19,000 were distinct, hashing to 4,900 (of 5,000 possible) different hash codes. The resulting inverted file indexes used 845,000 bytes, or about 2.6% of the size of the original files. The particularly small indexes are caused by the fact that keys are taken from only the first 50 non-common words of some very long input files.

Even this large *lookall* index can be searched quickly. For example, to find this document by looking for the keys "lesk inverted indexes" required 1.7 seconds of processor time and system time. By comparison, just to search the 800,000 byte dictionary (smaller than even the inverted indexes, let alone the 32,000,000 bytes of text files) with *grep* takes 29 seconds of processor time. The *lookall* program is thus useful when looking for a document which you believe is stored on-line, but do not know where. For example, many memos from the Computing Science Research Center are in its UNIX file system, but it is often difficult to guess where a particular memo might be (it might have several authors, each with many directories, and have been worked on by a secretary with yet more directories). Instructions for the use of the *lookall* command are given in the manual section, shown in the appendix to this memorandum.

The only indexes maintained routinely are those of publication lists and all English files. To make other indexes, the programs for making keys, sorting them, searching the indexes, and delivering answers must be used. Since they are usually invoked as parts of higher-level commands, they are not in the default command directory, but are available to any user in the

directory */usr/lib/refer*. Three programs are of interest: *mkey*, which isolates keys from input files; *inv*, which makes an index from a set of keys; and *hunt*, which searches the index and delivers the items. Note that the two parts of the retrieval phase are combined into one program, to avoid the excessive system work and delay which would result from running these as separate processes.

These three commands have a large number of options to adapt to different kinds of input. The user not interested in the detailed description that now follows may skip to section 3, which describes the *refer* program, a packaged-up version of these tools specifically oriented towards formatting references.

Make Keys. The program *mkey* is the key-making program corresponding to step (1) in phase A. Normally, it reads its input from the file names given as arguments, and if there are no arguments it reads from the standard input. It assumes that blank lines in the input delimit separate items, for each of which a different line of keys should be generated. The lines of keys are written on the standard output. Keys are any alphanumeric string in the input not among the most frequent words in English and not entirely numeric (except that all-numeric strings are acceptable if they are between 1900 and 1999). In the output, keys are translated to lower case, and truncated to six characters in length; any associated punctuation is removed. The following flag arguments are recognized by *mkey:*

−c *name*	Name of file of common words; default is */usr/lib/eign*.
−f *name*	Read a list of files from *name* and take each as an input argument.
−i *chars*	Ignore all lines which begin with '%' followed by any character in *chars*.
−k*n*	Use at most *n* keys per input item.
−l*n*	Ignore items shorter than *n* letters long.
−n*m*	Ignore as a key any word in the first *m* words of the list of common English words. The default is 100.
−s	Remove the labels *(file:start,length)* from the output; just give the keys. Used when searching rather than indexing.
−w	Each whole file is a separate item; blank lines in files are irrelevant.

The normal arguments for indexing references are the defaults, which are *−c /usr/lib/eign*, *−n100*, and *−l3*. For searching, the *−s* option is also needed. When the big *lookall* index of all English files is run, the options are *−w*, *−k50*, and *−f (filelist)*. When running on textual input, the *mkey* program processes about 1000 English words per processor second. Unless the *−k* option is used (and the input files are long enough for it to take effect) the output of *mkey* is comparable in size to its input.

Hash and invert. The *inv* program computes the hash codes and writes the inverted files. It reads the output of *mkey* and writes the set of files described earlier in this section. It expects one argument, which is used as the base name for the three (or four) files to be written. Assuming an argument of *Index* (the default) the entry file is named *Index.ia*, the posting file *Index.ib*, the tag file *Index.ic*, and the key file (if present) *Index.id*. The *inv* program recognizes the following options:

−a	Append the new keys to a previous set of inverted files, making new files if there is no old set using the same base name.
−d	Write the optional key file. This is needed when you can not check for false drops by looking for the keys in the original inputs, i.e. when the key derivation procedure is complicated and the output keys are not words from the input files.
−h*n*	The hash table size is *n* (default 997); *n* should be prime. Making *n* bigger saves search time and spends disk space.

−i[u] *name* Take input from file *name*, instead of the standard input; if **u** is present *name* is unlinked when the sort is started. Using this option permits the sort scratch space to overlap the disk space used for input keys.

−n Make a completely new set of inverted files, ignoring previous files.

−p Pipe into the sort program, rather than writing a temporary input file. This saves disk space and spends processor time.

−v Verbose mode; print a summary of the number of keys which finished indexing.

About half the time used in *inv* is in the contained sort. Assuming the sort is roughly linear, however, a guess at the total timing for *inv* is 250 keys per second. The space used is usually of more importance: the entry file uses four bytes per possible hash (note the **−h** option), and the tag file around 15-20 bytes per item indexed. Roughly, the posting file contains one item for each key instance and one item for each possible hash code; the items are two bytes long if the tag file is less than 65336 bytes long, and the items are four bytes wide if the tag file is greater than 65536 bytes long. To minimize storage, the hash tables should be over-full; for most of the files indexed in this way, there is no other real choice, since the *entry* file must fit in memory.

Searching and Retrieving. The *hunt* program retrieves items from an index. It combines, as mentioned above, the two parts of phase (B): search and delivery. The reason why it is efficient to combine delivery and search is partly to avoid starting unnecessary processes, and partly because the delivery operation must be a part of the search operation in any case. Because of the hashing, the search part takes place in two stages: first items are retrieved which have the right hash codes associated with them, and then the actual items are inspected to determine false drops, i.e. to determine if anything with the right hash codes doesn't really have the right keys. Since the original item is retrieved to check on false drops, it is efficient to present it immediately, rather than only giving the tag as output and later retrieving the item again. If there were a separate key file, this argument would not apply, but separate key files are not common.

Input to *hunt* is taken from the standard input, one query per line. Each query should be in *mkey −s* output format; all lower case, no punctuation. The *hunt* program takes one argument which specifies the base name of the index files to be searched. Only one set of index files can be searched at a time, although many text files may be indexed as a group, of course. If one of the text files has been changed since the index, that file is searched with *fgrep;* this may occasionally slow down the searching, and care should be taken to avoid having many out of date files. The following option arguments are recognized by *hunt:*

−a Give all output; ignore checking for false drops.

−C*n* Coordination level *n;* retrieve items with not more than *n* terms of the input missing; default *C0*, implying that each search term must be in the output items.

−F[yn*d***]** "−Fy" gives the text of all the items found; "−Fn" suppresses them. "−F*d*" where *d* is an integer gives the text of the first *d* items. The default is *−Fy.*

−g Do not use *fgrep* to search files changed since the index was made; print an error comment instead.

−i *string* Take *string* as input, instead of reading the standard input.

−l *n* The maximum length of internal lists of candidate items is *n;* default 1000.

−o *string* Put text output ("−Fy") in *string;* of use *only* when invoked from another program.

−p Print hash code frequencies; mostly for use in optimizing hash table sizes.

−T[yn*d*] "−Ty" gives the tags of the items found; "−Tn" suppresses them. "−T*d*" where *d* is an integer gives the first *d* tags. The default is *−Tn*.

−t *string* Put tag output ("−Ty") in *string;* of use *only* when invoked from another program.

The timing of *hunt* is complex. Normally the hash table is overfull, so that there will be many false drops on any single term; but a multi-term query will have few false drops on all terms. Thus if a query is underspecified (one search term) many potential items will be examined and discarded as false drops, wasting time. If the query is overspecified (a dozen search terms) many keys will be examined only to verify that the single item under consideration has that key posted. The variation of search time with number of keys is shown in the table below. Queries of varying length were constructed to retrieve a particular document from the file of references. In the sequence to the left, search terms were chosen so as to select the desired paper as quickly as possible. In the sequence on the right, terms were chosen inefficiently, so that the query did not uniquely select the desired document until four keys had been used. The same document was the target in each case, and the final set of eight keys are also identical; the differences at five, six and seven keys are produced by measurement error, not by the slightly different key lists.

Efficient Keys				Inefficient Keys			
No. keys	Total drops (incl. false)	Retrieved Documents	Search time (seconds)	No. keys	Total drops (incl. false)	Retrieved Documents	Search time (seconds)
1	15	3	1.27	1	68	55	5.96
2	1	1	0.11	2	29	29	2.72
3	1	1	0.14	3	8	8	0.95
4	1	1	0.17	4	1	1	0.18
5	1	1	0.19	5	1	1	0.21
6	1	1	0.23	6	1	1	0.22
7	1	1	0.27	7	1	1	0.26
8	1	1	0.29	8	1	1	0.29

As would be expected, the optimal search is achieved when the query just specifies the answer; however, overspecification is quite cheap. Roughly, the time required by *hunt* can be approximated as 30 milliseconds per search key plus 75 milliseconds per dropped document (whether it is a false drop or a real answer). In general, overspecification can be recommended; it protects the user against additions to the data base which turn previously uniquely-answered queries into ambiguous queries.

The careful reader will have noted an enormous discrepancy between these times and the earlier quoted time of around 1.9 seconds for a search. The times here are purely for the search and retrieval: they are measured by running many searches through a single invocation of the *hunt* program alone. Usually, the UNIX command processor (the shell) must start both the *mkey* and *hunt* processes for each query, and arrange for the output of *mkey* to be fed to the *hunt* program. This adds a fixed overhead of about 1.7 seconds of processor time to any single search. Furthermore, remember that all these times are processor times: on a typical morning on our PDP 11/70 system, with about one dozen people logged on, to obtain 1 second of processor time for the search program took between 2 and 12 seconds of real time, with a median of 3.9 seconds and a mean of 4.8 seconds. Thus, although the work involved in a single search may be only 200 milliseconds, after you add the 1.7 seconds of startup processor time and then assume a 4:1 elapsed/processor time ratio, it will be 8 seconds before any response is printed.

3. Selecting and Formatting References for TROFF

The major application of the retrieval software is *refer*, which is a *troff* preprocessor like *eqn*.[3] It scans its input looking for items of the form

```
.[
imprecise citation
.]
```

where an imprecise citation is merely a string of words found in the relevant bibliographic citation. This is translated into a properly formatted reference. If the imprecise citation does not correctly identify a single paper (either selecting no papers or too many) a message is given. The data base of citations searched may be tailored to each system, and individual users may specify their own citation files. On our system, the default data base is accumulated from the publication lists of the members of our organization, plus about half a dozen personal bibliographies that were collected. The present total is about 4300 citations, but this increases steadily. Even now, the data base covers a large fraction of local citations.

For example, the reference for the *eqn* paper above was specified as

```
...
preprocessor like
.I eqn.
.[
kernighan cherry acm 1975
.]
It scans its input looking for items
...
```

This paper was itself printed using *refer*. The above input text was processed by *refer* as well as *tbl* and *troff* by the command

refer memo-file | tbl | troff −ms

and the reference was automatically translated into a correct citation to the ACM paper on mathematical typesetting.

The procedure to use to place a reference in a paper using *refer* is as follows. First, use the *lookbib* command to check that the paper is in the data base and to find out what keys are necessary to retrieve it. This is done by typing *lookbib* and then typing some potential queries until a suitable query is found. For example, had one started to find the *eqn* paper shown above by presenting the query

```
$ lookbib
kernighan cherry
(EOT)
```

lookbib would have found several items; experimentation would quickly have shown that the query given above is adequate. Overspecifying the query is of course harmless; it is even desirable, since it decreases the risk that a document added to the publication data base in the future will be retrieved in addition to the intended document. The extra time taken by even a grossly overspecified query is quite small. A particularly careful reader may have noticed that "acm" does not appear in the printed citation: we have supplemented some of the data base items with extra keywords, such as common abbreviations for journals or other sources, to aid in searching.

If the reference is in the data base, the query that retrieved it can be inserted in the text, between .[and .] brackets. If it is not in the data base, it can be typed into a private file of

---

3. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach.* **18**, pp.151-157 (March 1975).

references, using the format discussed in the next section, and then the **−p** option used to search this private file. Such a command might read (if the private references are called *myfile*)

refer −p myfile document | *tbl* | *eqn* | *troff −ms . . .*

where *tbl* and/or *eqn* could be omitted if not needed. The use of the *−ms* macros[4] or some other macro package, however, is essential. *Refer* only generates the data for the references; exact formatting is done by some macro package, and if none is supplied the references will not be printed.

By default, the references are numbered sequentially, and the *−ms* macros format references as footnotes at the bottom of the page. This memorandum is an example of that style. Other possibilities are discussed in section 5 below.

4. Reference Files.

A reference file is a set of bibliographic references usable with *refer*. It can be indexed using the software described in section 2 for fast searching. What *refer* does is to read the input document stream, looking for imprecise citation references. It then searches through reference files to find the full citations, and inserts them into the document. The format of the full citation is arranged to make it convenient for a macro package, such as the *−ms* macros, to format the reference for printing. Since the format of the final reference is determined by the desired style of output, which is determined by the macros used, *refer* avoids forcing any kind of reference appearance. All it does is define a set of string registers which contain the basic information about the reference; and provide a macro call which is expanded by the macro package to format the reference. It is the responsibility of the final macro package to see that the reference is actually printed; if no macros are used, and the output of *refer* fed untranslated to *troff*, nothing at all will be printed.

The strings defined by *refer* are taken directly from the files of references, which are in the following format. The references should be separated by blank lines. Each reference is a sequence of lines beginning with % and followed by a key-letter. The remainder of that line, and successive lines until the next line beginning with %, contain the information specified by the key-letter. In general, *refer* does not interpret the information, but merely presents it to the macro package for final formatting. A user with a separate macro package, for example, can add new key-letters or use the existing ones for other purposes without bothering *refer*.

The meaning of the key-letters given below, in particular, is that assigned by the *−ms* macros. Not all information, obviously, is used with each citation. For example, if a document is both an internal memorandum and a journal article, the macros ignore the memorandum version and cite only the journal article. Some kinds of information are not used at all in printing the reference; if a user does not like finding references by specifying title or author keywords, and prefers to add specific keywords to the citation, a field is available which is searched but not printed (**K**).

The key letters currently recognized by *refer* and *−ms,* with the kind of information implied, are:

---

4. M. E. Lesk, *Typing Documents on UNIX and GCOS: The -ms Macros for Troff,* Bell Laboratories internal memorandum (1977).

Key	Information specified	Key	Information specified
A	Author's name	N	Issue number
B	Title of book containing item	O	Other information
C	City of publication	P	Page(s) of article
D	Date	R	Technical report reference
E	Editor of book containing item	T	Title
G	Government (NTIS) ordering number	V	Volume number
I	Issuer (publisher)		
J	Journal name		
K	Keys (for searching)	X	or
L	Label	Y	or
M	Memorandum label	Z	Information not used by *refer*

For example, a sample reference could be typed as:

```
%T Bounds on the Complexity of the Maximal
Common Subsequence Problem
%Z ctr127
%A A. V. Aho
%A D. S. Hirschberg
%A J. D. Ullman
%J J. ACM
%V 23
%N 1
%P 1-12
%M abcd-78
%D Jan. 1976
```

Order is irrelevant, except that authors are shown in the order given. The output of *refer* is a stream of string definitions, one for each of the fields of each reference, as shown below.

```
.]-
.ds [A authors' names ...
.ds [T title ...
.ds [J journal ...
...
.] [ type-number
```

The *refer* program, in general, does not concern itself with the significance of the strings. The different fields are treated identically by *refer*, except that the X, Y and Z fields are ignored (see the **−i** option of *mkey*) in indexing and searching. All *refer* does is select the appropriate citation, based on the keys. The macro package must arrange the strings so as to produce an appropriately formatted citation. In this process, it uses the convention that the 'T' field is the title, the 'J' field the journal, and so forth.

The *refer* program does arrange the citation to simplify the macro package's job, however. The special macro **.]−** precedes the string definitions and the special macro **.][** follows. These are changed from the input **.[** and **.]** so that running the same file through *refer* again is harmless. The **.]−** macro can be used by the macro package to initialize. The **.][** macro, which should be used to print the reference, is given an argument *type-number* to indicate the kind of reference, as follows:

Value	Kind of reference
1	Journal article
2	Book
3	Article within book
4	Technical report
5	Bell Labs technical memorandum
0	Other

The type is determined by the presence or absence of particular fields in the citation (a journal article must have a 'J' field, a book must have an 'I' field, and so forth). To a small extent, this violates the above rule that *refer* does not concern itself with the contents of the citation; however, the classification of the citation in *troff* macros would require a relatively expensive and obscure program. Any macro writer may, of course, preserve consistency by ignoring the argument to the **.][** macro.

The reference is flagged in the text with the sequence

```
\*([.number\*(.]
```

where *number* is the footnote number. The strings **[.** and **.]** should be used by the macro package to format the reference flag in the text. These strings can be replaced for a particular footnote, as described in section 5. The footnote number (or other signal) is available to the reference macro **.][** as the string register **[F**. To simplify dealing with a text reference that occurs at the end of a sentence, *refer* treats a reference which follows a period in a special way. The period is removed, and the reference is preceded by a call for the string **<.** and followed by a call for the string **>.** For example, if a reference follows "end." it will appear as

```
end\*(<.\*([.number\*(.]\*(>.
```

where *number* is the footnote number. The macro package should turn either the string **>.** or **<.** into a period and delete the other one. This permits the output to have either the form "end[31]." or "end.^{31}" as the macro package wishes. Note that in one case the period precedes the number and in the other it follows the number.

In some cases users wish to suspend the searching, and merely use the reference macro formatting. That is, the user doesn't want to provide a search key between **.[** and **.]** brackets, but merely the reference lines for the appropriate document. Alternatively, the user can wish to add a few fields to those in the reference as in the standard file, or override some fields. Altering or replacing fields, or supplying whole references, is easily done by inserting lines beginning with **%**; any such line is taken as direct input to the reference processor rather than keys to be searched. Thus

```
.[
key1 key2 key3 ...
%Q New format item
%R Override report name
.]
```

makes the indicates changes to the result of searching for the keys. All of the search keys must be given before the first **%** line.

If no search keys are provided, an entire citation can be provided in-line in the text. For example, if the *eqn* paper citation were to be inserted in this way, rather than by searching for it in the data base, the input would read

```
...
preprocessor like
.I eqn.
.[
%A B. W. Kernighan
%A L. L. Cherry
%T A System for Typesetting Mathematics
%J Comm. ACM
%V 18
%N 3
%P 151-157
%D March 1975
.]
It scans its input looking for items
...
```

This would produce a citation of the same appearance as that resulting from the file search.

As shown, fields are normally turned into *troff* strings. Sometimes users would rather have them defined as macros, so that other *troff* commands can be placed into the data. When this is necessary, simply double the control character % in the data. Thus the input

```
.[
%V 23
%%M
Bell Laboratories,
Murray Hill, N.J. 07974
.]
```

is processed by *refer* into

```
.ds [V 23
.de [M
Bell Laboratories,
Murray Hill, N.J. 07974
..
```

The information after **%%M** is defined as a macro to be invoked by **.[M** while the information after **%V** is turned into a string to be invoked by *([V. At present *−ms* expects all information as strings.

5. Collecting References and other Refer Options

Normally, the combination of *refer* and *−ms* formats output as *troff* footnotes which are consecutively numbered and placed at the bottom of the page. However, options exist to place the references at the end; to arrange references alphabetically by senior author; and to indicate references by strings in the text of the form [Name1975a] rather than by number. Whenever references are not placed at the bottom of a page identical references are coalesced.

For example, the −e option to *refer* specifies that references are to be collected; in this case they are output whenever the sequence

```
.[
$LIST$
.]
```

is encountered. Thus, to place references at the end of a paper, the user would run *refer* with the *−e* option and place the above $LIST$ commands after the last line of the text. *Refer* will then move all the references to that point. To aid in formatting the collected references, *refer* writes the references preceded by the line

.]<

and followed by the line

.]>

to invoke special macros before and after the references.

Another possible option to *refer* is the **−s** option to specify sorting of references. The default, of course, is to list references in the order presented. The **−s** option implies the **−e** option, and thus requires a

.[
$LIST$.
.]

entry to call out the reference list. The **−s** option may be followed by a string of letters, numbers, and '+' signs indicating how the references are to be sorted. The sort is done using the fields whose key-letters are in the string as sorting keys; the numbers indicate how many of the fields are to be considered, with '+' taken as a large number. Thus the default is **−sAD** meaning "Sort on senior author, then date." To sort on all authors and then title, specify **−sA+T**. And to sort on two authors and then the journal, write **−sA2J**.

Other options to *refer* change the signal or label inserted in the text for each reference. Normally these are just sequential numbers, and their exact placement (within brackets, as superscripts, etc.) is determined by the macro package. The **−l** option replaces reference numbers by strings composed of the senior author's last name, the date, and a disambiguating letter. If a number follows the **l** as in **−l3** only that many letters of the last name are used in the label string. To abbreviate the date as well the form **-l***m,n* shortens the last name to the first *m* letters and the date to the last *n* digits. For example, the option **−l3,2** would refer to the *eqn* paper (reference 3) by the signal *Ker75a*, since it is the first cited reference by Kernighan in 1975.

A user wishing to specify particular labels for a private bibliography may use the **−k** option. Specifying **−k***x* causes the field *x* to be used as a label. The default is **L**. If this field ends in **−**, that character is replaced by a sequence letter; otherwise the field is used exactly as given.

If none of the *refer*-produced signals are desired, the **−b** option entirely suppresses automatic text signals.

If the user wishes to override the *−ms* treatment of the reference signal (which is normally to enclose the number in brackets in *nroff* and make it a superscript in *troff*) this can be done easily. If the lines .[or .] contain anything following these characters, the remainders of these lines are used to surround the reference signal, instead of the default. Thus, for example, to say "See reference (2)." and avoid "See reference.[2]" the input might appear

See reference
.[(
imprecise citation ...
.]).

Note that blanks are significant in this construction. If a permanent change is desired in the style of reference signals, however, it is probably easier to redefine the strings [. and .] (which are used to bracket each signal) than to change each citation.

Although normally *refer* limits itself to retrieving the data for the reference, and leaves to a macro package the job of arranging that data as required by the local format, there are two special options for rearrangements that can not be done by macro packages. The **−c** option puts fields into all upper case (CAPS-SMALL CAPS in *troff* output). The key-letters indicated what information is to be translated to upper case follow the **c**, so that **−cAJ** means that authors' names and journals are to be in caps. The **−a** option writes the names of authors last

name first, that is *A. D. Hall, Jr.* is written as *Hall, A. D. Jr.* The citation form of the *Journal of the ACM*, for example, would require both **−cA** and **−a** options. This produces authors' names in the style *KERNIGHAN, B. W. AND CHERRY, L. L.* for the previous example. The **−a** option may be followed by a number to indicate how many author names should be reversed; **−a1** (without any **−c** option) would produce *Kernighan, B. W. and L. L. Cherry,* for example.

Finally, there is also the previously-mentioned **−p** option to let the user specify a private file of references to be searched before the public files. Note that *refer* does not insist on a previously made index for these files. If a file is named which contains reference data but is not indexed, it will be searched (more slowly) by *refer* using *fgrep.* In this way it is easy for users to keep small files of new references, which can later be added to the public data bases.

CHAPTER 5

COMMAND REFERENCE

Included in this chapter are the _XENIX Programmer's Manual_ manual pages for commands discussed in this manual. They have been included here for completeness.

NAME

intro - introduction to commands

DESCRIPTION

This section describes publicly accessible commands in alphabetic order. Certain distinctions of purpose are made in the headings:

(1) Commands of general utility.

(1C) Commands for communication with other systems.

(1G) Commands used primarily for graphics and computer-aided design.

(1M) Commands used primarily for system maintenance.

The word `local' at the foot of a page means that the command is not intended for general distribution.

SEE ALSO
DIAGNOSTICS

Section (6) for computer games.

How to get started, in the Introduction.

DIAGNOSTICS

Upon termination each command returns two bytes of status, one supplied by the system giving the cause for termination, and (in the case of `normal' termination) one supplied by the program, see wait and exit(2). The former byte is 0 for normal termination, the latter is customarily 0 for successful execution, nonzero to indicate troubles such as erroneous parameters, bad or inaccessible data, or other inability to cope with the task at hand. It is called variously `exit code', `exit status' or `return code', and is described only where special conventions are involved.

NAME

awk - pattern scanning and processing language

SYNTAX

awk [-F*c*] [prog] [file] ...

DESCRIPTION

Awk scans each input *file* for lines that match any of a set of patterns specified in *prog*. With each pattern in *prog* there can be an associated action that will be performed when a line of a *file* matches the pattern. The set of patterns may appear literally as *prog*, or in a file specified as -f *file*.

Files are read in order; if there are no files, the standard input is read. The file name `-' means the standard input. Each line is matched against the pattern portion of every pattern-action statement; the associated action is performed for each matched pattern.

An input line is made up of fields separated by white space. (This default can be changed by using FS, *vide infra*.) The fields are denoted $1, $2, ... ; $0 refers to the entire line.

A pattern-action statement has the form

```
pattern { action }
```

A missing { action } means print the line; a missing pattern always matches.

An action is a sequence of statements. A statement can be one of the following:

```
if ( conditional ) statement [ else statement ]
while ( conditional ) statement
for ( expression ; conditional ; expression ) statement
break
continue
{ [ statement ] ... }
variable = expression
print [ expression-list ] [ >expression ]
printf format [ , expression-list ] [ >expression ]
next # skip remaining patterns on this input line
exit # skip the rest of the input
```

Statements are terminated by semicolons, newlines or right braces. An empty expression-list stands for the whole line. Expressions take on string or numeric values as appropriate, and are built using the operators +, -, *, /, %, and concatenation (indicated by a blank). The C operators ++, --,

+=, -=, *=, /=, and %= are also available in expressions. Variables may be scalars, array elements (denoted x[i]) or fields. Variables are initialized to the null string. Array subscripts may be any string, not necessarily numeric; this allows for a form of associative memory. String constants are quoted "...".

The _print_ statement prints its arguments on the standard output (or on a file if >_file_ is present), separated by the current output field separator, and terminated by the output record separator. The _printf_ statement formats its expression list according to the format (see _printf_(3)).

The built-in function _length_ returns the length of its argument taken as a string, or of the whole line if no argument. There are also built-in functions _exp_, _log_, _sqrt_, and _int_. The last truncates its argument to an integer. _substr_(_s_, _m_, _n_) returns the _n_-character substring of _s_ that begins at position _m_. The function _sprintf_(_fmt_, _expr_, _expr_, ...) formats the expressions according to the _printf_(3) format given by _fmt_ and returns the resulting string.

Patterns are arbitrary Boolean combinations (!, ||, &&, and parentheses) of regular expressions and relational expressions. Regular expressions must be surrounded by slashes and are as in _egrep_. Isolated regular expressions in a pattern apply to the entire line. Regular expressions may also occur in relational expressions.

A pattern may consist of two patterns separated by a comma; in this case, the action is performed for all lines between an occurrence of the first pattern and the next occurrence of the second.

A relational expression is one of the following:

    expression matchop regular-expression
    expression relop expression

where a relop is any of the six relational operators in C, and a matchop is either ~ (for contains) or !~ (for does not contain). A conditional is an arithmetic expression, a relational expression, or a Boolean combination of these.

The special patterns BEGIN and END may be used to capture control before the first input line is read and after the last. BEGIN must be the first pattern, END the last.

A single character _c_ may be used to separate the fields by starting the program with

```
BEGIN { FS = "c" }
```

or by using the -F*c* option.

Other variable names with special meanings include NF, the number of fields in the current record; NR, the ordinal number of the current record; FILENAME, the name of the current input file; OFS, the output field separator (default blank); ORS, the output record separator (default newline); and OFMT, the output format for numbers (default "%.6g").

EXAMPLES

Print lines longer than 72 characters:

```
length > 72
```

Print first two fields in opposite order:

```
{ print $2, $1 }
```

Add up first column, print sum and average:

```
    { s += $1 }
END { print "sum is", s, " average is", s/NR }
```

Print fields in reverse order:

```
{ for (i = NF; i > 0; --i) print $i }
```

Print all lines between start/stop pairs:

```
/start/, /stop/
```

Print all lines whose first field is different from previous one:

```
$1 != prev { print; prev = $1 }
```

SEE ALSO

lex(1), sed(1)
A. V. Aho, B. W. Kernighan, P. J. Weinberger, *Awk - a pattern scanning and processing language*

NOTES

There are no explicit conversions between numbers and strings. To force an expression to be treated as a number add 0 to it; to force it to be treated as a string concatenate "" to it.

NAME

col - filter reverse line feeds

SYNTAX

col [-bfx]

DESCRIPTION

Col reads the standard input and writes the standard output. It performs the line overlays implied by reverse line feeds (ESC-7 in ASCII) and by forward and reverse half line feeds (ESC-9 and ESC-8). *Col* is particularly useful for filtering multicolumn output made with the `.rt' command of *nroff* and output resulting from use of the *tbl*(1) preprocessor.

Although *col* accepts half line motions in its input, it normally does not emit them on output. Instead, text that would appear between lines is moved to the next lower full line boundary. This treatment can be suppressed by the **-f** (fine) option; in this case the output from *col* may contain forward half line feeds (ESC-9), but will still never contain either kind of reverse line motion.

If the **-b** option is given, *col* assumes that the output device in use is not capable of backspacing. In this case, if several characters are to appear in the same place, only the last one read will be taken.

The control characters SO (ASCII code 017), and SI (016) are assumed to start and end text in an alternate character set. The character set (primary or alternate) associated with each printing character read is remembered; on output, SO and SI characters are generated where necessary to maintain the correct treatment of each character.

Col normally converts white space to tabs to shorten printing time. If the **-x** option is given, this conversion is suppressed.

All control characters are removed from the input except space, backspace, tab, return, newline, ESC (033) followed by one of 789, SI, SO, and VT (013). This last character is an alternate form of full reverse line feed, for compatibility with some other hardware conventions. All other non-printing characters are ignored.

SEE ALSO

troff(1), tbl(1), greek(1)

NOTES

Can't back up more than 128 lines.
No more than 800 characters, including backspaces, on a line.

NAME

comm - select or reject lines common to two sorted files

SYNTAX

comm [- [**123**]] file1 file2

DESCRIPTION

Comm reads *file1* and *file2*, which should be ordered in ASCII collating sequence, and produces a three column output: lines only in *file1*; lines only in *file2*; and lines in both files. The filename `-' means the standard input.

Flags 1, 2, or 3 suppress printing of the corresponding column. Thus **comm -12** prints only the lines common to the two files; **comm -23** prints only lines in the first file but not in the second; **comm -123** is a no-op.

SEE ALSO

cmp(1), diff(1), uniq(1)

NAME

ctags - create a tags file

SYNTAX

ctags [**-u**] [**-v**] [**-w**] [**-x**] name ...

DESCRIPTION

Ctags makes a tags file for ex(1) from the specified C, Pascal and Fortran sources. A tags file gives the locations of specified objects (in this case functions) in a group of files. Each line of the tags file contains the function name, the file in which it is defined, and a scanning pattern used to find the function definition. These are given in separate fields on the line, separated by blanks or tabs. Using the tags file, ex can quickly find these function definitions.

If the **-x** flag is given, ctags produces a list of function names, the line number and file name on which each is defined, as well as the text of that line and prints this on the standard output. This is a simple index which can be printed out as an off-line readable function index.

If the **-v** flag is given, an index of the form expected by vgrind(1) is produced on the standard output. This listing contains the function name, file name, and page number (assuming 64 line pages). Since the output will be sorted into lexicographic order, it may be desired to run the output through **sort -f**. Sample use:

```
ctags -v files | sort -f > index
vgrind -x index
```

Files whose name ends in **.c** or **.h** are assumed to be C source files and are searched for C routine and macro definitions. Others are first examined to see if they contain any Pascal or Fortran routine definitions; if not, they are processed again looking for C definitions.

Other options are:

-w suppressing warning diagnostics.

-u causing the specified files to be updated in tags, that is, all references to them are deleted, and the new values are appended to the file. (Beware: this option is implemented in a way which is rather slow; it is usually faster to simply rebuild the tags file.)

The tag main is treated specially in C programs. The tag formed is created by prepending M to the name of the file, with a trailing .c removed, if any, and leading pathname components also removed. This makes use of ctags practical

in directories with more than one program.

FILES

tags output tags file

SEE ALSO

ex(1), vi(1)

AUTHOR

Ken Arnold; FORTRAN added by Jim Kleckner; Bill Joy added Pascal and **-x**, replacing cxref.

NOTES

Recognition of **functions, subroutines** and **procedures** for FORTRAN and Pascal is done is a very simpleminded way. No attempt is made to deal with block structure; if you have two Pascal procedures in different blocks with the same name you lose.

The method of deciding whether to look for C or Pascal and FORTRAN functions is a hack.

NAME
deroff - remove nroff, troff, tbl and eqn constructs

SYNTAX
deroff [-w] file ...

DESCRIPTION
Deroff reads each file in sequence and removes all *nroff* and *troff* command lines, backslash constructions, macro definitions, *eqn* constructs (between `.EQ' and `.EN' lines or between delimiters), and table descriptions and writes the remainder on the standard output. *Deroff* follows chains of included files (`.so' and `.nx' commands); if a file has already been included, a `.so' is ignored and a `.nx' terminates execution. If no input file is given, *deroff* reads from the standard input file.

If the -w flag is given, the output is a word list, one `word' (string of letters, digits, and apostrophes, beginning with a letter; apostrophes are removed) per line, and all other characters ignored. Otherwise, the output follows the original, with the deletions mentioned above.

SEE ALSO
troff(1), eqn(1), tbl(1)

NOTES
Deroff is not a complete *troff* interpreter, so it can be confused by subtle constructs. Most errors result in too much rather than too little output.

NAME

diff - differential file comparator

SYNTAX

diff [-efbh] file1 file2

DESCRIPTION

Diff tells what lines must be changed in two files to bring them into agreement. If *file1* (*file2*) is `-', the standard input is used. If *file1* (*file2*) is a directory, then a file in that directory whose file-name is the same as the file-name of *file2* (*file1*) is used. The normal output contains lines of these forms:

n1 a *n3*,*n4*
n1,*n2* d *n3*
n1,*n2* c *n3*,*n4*

These lines resemble *ed* commands to convert *file1* into *file2*. The numbers after the letters pertain to *file2*. In fact, by exchanging `a' for `d' and reading backward one may ascertain equally how to convert *file2* into *file1*. As in *ed*, identical pairs where *n1* = *n2* or *n3* = *n4* are abbreviated as a single number.

Following each of these lines come all the lines that are affected in the first file flagged by `<', then all the lines that are affected in the second file flagged by `>'.

The -b option causes trailing blanks (spaces and tabs) to be ignored and other strings of blanks to compare equal.

The -e option produces a script of *a*, *c* and *d* commands for the editor *ed*, which will recreate *file2* from *file1*. The -f option produces a similar script, not useful with *ed*, in the opposite order. In connection with -e, the following shell program may help maintain multiple versions of a file. Only an ancestral file ($1) and a chain of version-to-version *ed* scripts ($2,$3,...) made by *diff* need be on hand. A `latest version' appears on the standard output.

```
(shift; cat $*; echo '1,$p') | ed - $1
```

Except in rare circumstances, *diff* finds a smallest sufficient set of file differences.

Option -h does a fast, half-hearted job. It works only when changed stretches are short and well separated, but does work on files of unlimited length. Options -e and -f are unavailable with -h.

FILES

/tmp/d?????
/usr/lib/diffh for **-h**

SEE ALSO

cmp(1), comm(1), ed(1)

DIAGNOSTICS

Exit status is 0 for no differences, 1 for some, 2 for trouble.

NOTES

Editing scripts produced under the **-e** or **-f** option are naive about creating lines consisting of a single `.'.

NAME

diff3 - 3-way differential file comparison

SYNTAX

diff3 [**-ex3**] file1 file2 file3

DESCRIPTION

Diff3 compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

====	all three files differ
====1	_file1_ is different
====2	_file2_ is different
====3	_file3_ is different

The type of change suffered in converting a given range of a given file to some other is indicated in one of these ways:

_f_ : _n1_ **a**	Text is to be appended after line number _n1_ in file _f_, where _f_ = 1, 2, or 3.
_f_ : _n1_ , _n2_ **c**	Text is to be changed in the range line _n1_ to line _n2_. If _n1_ = _n2_, the range may be abbreviated to _n1_.

The original contents of the range follows immediately after a **c** indication. When the contents of two files are identical, the contents of the lower-numbered file is suppressed.

Under the -e option, _diff3_ publishes a script for the editor _ed_ that will incorporate into _file1_ all changes between _file2_ and _file3_, _i.e_. the changes that normally would be flagged ==== and ====3. Option **-x** (-3) produces a script to incorporate only changes flagged ==== (====3). The following command will apply the resulting script to `file1'.

```
(cat script; echo '1,$p') | ed - file1
```

FILES

/tmp/d3?????
/usr/lib/diff3

SEE ALSO

diff(1)

NOTES

Text lines that consist of a single `.' will defeat -e.
Files longer than 64K bytes won't work.

NAME

ed - text editor

SYNTAX

ed [**-**] [**-x**] [name]

DESCRIPTION

Ed is the standard text editor.

If a name argument is given, ed simulates an e command (see below) on the named file; that is to say, the file is read into ed's buffer so that it can be edited. If **-x** is present, an x command is simulated first to handle an encrypted file. The optional - suppresses the printing of character counts by e, r, and w commands.

Ed operates on a copy of any file it is editing; changes made in the copy have no effect on the file until a w (write) command is given. The copy of the text being edited resides in a temporary file called the buffer.

Commands to ed have a simple and regular structure: zero or more addresses followed by a single character command, possibly followed by parameters to the command. These addresses specify one or more lines in the buffer. Missing addresses are supplied by default.

In general, only one command may appear on a line. Certain commands allow the addition of text to the buffer. While ed is accepting text, it is said to be in input mode. In this mode, no commands are recognized; all input is merely collected. Input mode is left by typing a period `.' alone at the beginning of a line.

Ed supports a limited form of regular expression notation. A regular expression specifies a set of strings of characters. A member of this set of strings is said to be matched by the regular expression. In the following specification for regular expressions the word `character' means any character but newline.

1. Any character except a special character matches itself. Special characters are the regular expression delimiter plus \[. and sometimes ^*$.

2. A . matches any character.

3. A \ followed by any character except a digit or () matches that character.

4. A nonempty string s bracketed [s] (or [^s]) matches any character in (or not in) s. In s, \ has no special

meaning, and] may only appear as the first letter. A substring a-b, with a and b in ascending ASCII order, stands for the inclusive range of ASCII characters.

5. A regular expression of form 1-4 followed by * matches a sequence of 0 or more matches of the regular expression.

6. A regular expression, x, of form 1-8, bracketed \(x\) matches what x matches.

7. A \ followed by a digit n matches a copy of the string that the bracketed regular expression beginning with the nth \(matched.

8. A regular expression of form 1-8, x, followed by a regular expression of form 1-7, y matches a match for x followed by a match for y, with the x match being as long as possible while still permitting a y match.

9. A regular expression of form 1-8 preceded by ^ (or followed by $), is constrained to matches that begin at the left (or end at the right) end of a line.

10. A regular expression of form 1-9 picks out the longest among the leftmost matches in a line.

11. An empty regular expression stands for a copy of the last regular expression encountered.

Regular expressions are used in addresses to specify lines and in one command (see s below) to specify a portion of a line which is to be replaced. If it is desired to use one of the regular expression metacharacters as an ordinary character, that character may be preceded by `\'. This also applies to the character bounding the regular expression (often `/') and to `\' itself.

To understand addressing in ed it is necessary to know that at any time there is a current line. Generally speaking, the current line is the last line affected by a command; however, the exact effect on the current line is discussed under the description of the command. Addresses are constructed as follows.

1. The character `.' addresses the current line.

2. The character `$' addresses the last line of the buffer.

3. A decimal number n addresses the n-th line of the buffer.

4. `'x' addresses the line marked with the name x, which must be a lower-case letter. Lines are marked with the k command described below.

5. A regular expression enclosed in slashes `/' addresses the line found by searching forward from the current line and stopping at the first line containing a string that matches the regular expression. If necessary the search wraps around to the beginning of the buffer.

6. A regular expression enclosed in queries `?' addresses the line found by searching backward from the current line and stopping at the first line containing a string that matches the regular expression. If necessary the search wraps around to the end of the buffer.

7. An address followed by a plus sign `+' or a minus sign `-' followed by a decimal number specifies that address plus (resp. minus) the indicated number of lines. The plus sign may be omitted.

8. If an address begins with `+' or `-' the addition or subtraction is taken with respect to the current line; e.g. `-5' is understood to mean `.-5'.

9. If an address ends with `+' or `-', then 1 is added (resp. subtracted). As a consequence of this rule and rule 8, the address `-' refers to the line before the current line. Moreover, trailing `+' and `-' characters have cumulative effect, so `--' refers to the current line less 2.

10. To maintain compatibility with earlier versions of the editor, the character `^' in addresses is equivalent to `-'.

Commands may require zero, one, or two addresses. Commands which require no addresses regard the presence of an address as an error. Commands which accept one or two addresses assume default addresses when insufficient are given. If more addresses are given than such a command requires, the last one or two (depending on what is accepted) are used.

Addresses are separated from each other typically by a comma `,'. They may also be separated by a semicolon `;'. In this case the current line `.' is set to the previous address before the next address is interpreted. This feature can be used to determine the starting line for forward and backward searches (`/', `?'). The second address of any two-address sequence must correspond to a line following the line corresponding to the first address.

In the following list of ed commands, the default addresses are shown in parentheses. The parentheses are not part of the address, but are used to show that the given addresses are the default.

As mentioned, it is generally illegal for more than one command to appear on a line. However, most commands may be suffixed by `p' or by `l', in which case the current line is either printed or listed respectively in the way discussed below.

(.)a
<text>
.

The append command reads the given text and appends it after the addressed line. `.' is left on the last line input, if there were any, otherwise at the addressed line. Address `0' is legal for this command; text is placed at the beginning of the buffer.

(., .)c
<text>
.

The change command deletes the addressed lines, then accepts input text which replaces these lines. `.' is left at the last line input; if there were none, it is left at the line preceding the deleted lines.

(., .)d

The delete command deletes the addressed lines from the buffer. The line originally after the last line deleted becomes the current line; if the lines deleted were originally at the end, the new last line becomes the current line.

e filename

The edit command causes the entire contents of the buffer to be deleted, and then the named file to be read in. `.' is set to the last line of the buffer. The number of characters read is typed. `filename' is remembered for possible use as a default file name in a subsequent r or w command. If `filename' is missing, the remembered name is used.

E filename

This command is the same as e, except that no diagnostic results when no w has been given since the last buffer alteration.

f filename

The filename command prints the currently remembered file name. If `filename' is given, the currently

remembered file name is changed to `filename'.

(1,$)g/regular expression/command list
In the global command, the first step is to mark every line which matches the given regular expression. Then for every such line, the given command list is executed with `.' initially set to that line. A single command or the first of multiple commands appears on the same line with the global command. All lines of a multi-line list except the last line must be ended with `\'. A, i, and c commands and associated input are permitted; the `.' terminating input mode may be omitted if it would be on the last line of the command list. The commands g and v are not permitted in the command list.

(.)i

<text>

.

This command inserts the given text before the addressed line. `.' is left at the last line input, or, if there were none, at the line before the addressed line. This command differs from the a command only in the placement of the text.

(., .+1)j
This command joins the addressed lines into a single line; intermediate newlines simply disappear. `.' is left at the resulting line.

(.)kx
The mark command marks the addressed line with name x, which must be a lower-case letter. The address form `'x' then addresses this line.

(., .)l
The list command prints the addressed lines in an unambiguous way: non-graphic characters are printed in two-digit octal, and long lines are folded. The l command may be placed on the same line after any non-i/o command.

(., .)ma
The move command repositions the addressed lines after the line addressed by a. The last of the moved lines becomes the current line.

(., .)p
The print command prints the addressed lines. `.' is left at the last line printed. The p command may be placed on the same line after any non-i/o command.

(., .)P
This command is a synonym for *p*.

q The quit command causes *ed* to exit. No automatic write of a file is done.

Q This command is the same as *q*, except that no diagnostic results when no *w* has been given since the last buffer alteration.

($)r filename
The read command reads in the given file after the addressed line. If no file name is given, the remembered file name, if any, is used (see *e* and *f* commands). The file name is remembered if there was no remembered file name a gl
r and causes the file to be read at the beginning of the buffer. If the read is successful, the number of characters read is typed. `.' is left at the last line read in from the file.

(., .)s/regular expression/replacement/ or,
(., .)s/regular expression/replacement/g
The substitute command searches each addressed line for an occurrence of the specified regular expression. On each line in which a match is found, all matched strings are replaced by the replacement specified, if the global replacement indicator `g' appears after the command. If the global indicator does not appear, only the first occurrence of the matched string is replaced. It is an error for the substitution to fail on all addressed lines. Any character other than space or new-line may be used instead of `/' to delimit the regular expression and the replacement. `.' is left at the last line substituted.

An ampersand `&' appearing in the replacement is replaced by the string matching the regular expression. The special meaning of `&' in this context may be suppressed by preceding it by `\'. The characters `*n*' where *n* is a digit, are replaced by the text matched by the *n*-th regular subexpression enclosed between `\(' and `\)'. When nested, parenthesized subexpressions are present, *n* is determined by counting occurrences of `\(' starting from the left.

Lines may be split by substituting new-line characters into them. The new-line in the replacement string must be escaped by preceding it by `\'.

(., .)t*a*
This command acts just like the *m* command, except that

a copy of the addressed lines is placed after address *a* (which may be 0). `.' is left on the last line of the copy.

(., .)u
The undo command restores the preceding contents of the current line, which must be the last line in which a substitution was made.

(1, $)v/regular expression/command list
This command is the same as the global command *g* except that the command list is executed *g* with `.' initially set to every line *except* those matching the regular expression.

(1, $)w filename
The write command writes the addressed lines onto the given file. If the file does not exist, it is created mode 666 (readable and writable by everyone). The file name is remembered if there was no remembered file name already. If no file name is given, the remembered file name, if any, is used (see *e* and *f* commands). `.' is unchanged. If the command is successful, the number of characters written is printed.

(1,$)W filename
This command is the same as *w*, except that the addressed lines are appended to the file.

x A key string is demanded from the standard input. Later *r*, *e* and *w* commands will encrypt and decrypt the text with this key by the algorithm of *crypt*(1). An explicitly empty key turns off encryption.

($)= The line number of the addressed line is typed. `.' is unchanged by this command.

!<shell command>
The remainder of the line after the `!' is sent to *sh*(1) to be interpreted as a command. `.' is unchanged.

(.+1)<newline>
An address alone on a line causes the addressed line to be printed. A blank line alone is equivalent to `.+1p'; it is useful for stepping through text.

If an interrupt signal (ASCII DEL) is sent, *ed* prints a `?' and returns to its command level.

Some size limitations: 512 characters per line, 256 characters per global command list, 64 characters per file name,

and 128K characters in the temporary file. The limit on the number of lines depends on the amount of core: each line takes 1 word.

When reading a file, _ed_ discards ASCII NUL characters and all characters after the last newline. It refuses to read files containing non-ASCII characters.

FILES

/tmp/e*
ed.hup: work is saved here if terminal hangs up

SEE ALSO

B. W. Kernighan, _A Tutorial Introduction to the ED Text Editor_
B. W. Kernighan, _Advanced editing on UNIX_
sed(1), crypt(1)

DIAGNOSTICS

\`?name' for inaccessible file; \`?' for errors in commands; \`?TMP' for temporary file overflow.

To protect against throwing away valuable work, a _q_ or _e_ command is considered to be in error, unless a _w_ has occurred since the last buffer change. A second _q_ or _e_ will be obeyed regardless.

NOTES

The _l_ command mishandles DEL.
A ! command cannot be subject to a _g_ command.
Because 0 is an illegal address for a _w_ command, it is not possible to create an empty file with _ed_. delim $$

NAME

eqn, neqn, checkeq - typeset mathematics

SYNTAX

eqn [-dxy] [-pn] [-sn] [-fn] [file] ...
checkeq [file] ...

DESCRIPTION

Eqn is a troff(1) preprocessor for typesetting mathematics on a Graphic Systems phototypesetter, *neqn* on terminals. Usage is almost always

```
eqn file ... | troff
neqn file ... | nroff
```

If no files are specified, these programs reads from the standard input. A line beginning with `.EQ' marks the start of an equation; the end of an equation is marked by a line beginning with `.EN'. Neither of these lines is altered, so they may be defined in macro packages to get centering, numbering, etc. It is also possible to set two characters as `delimiters'; subsequent text between delimiters is also treated as *eqn* input. Delimiters may be set to characters *x* and *y* with the command-line argument -d*xy* or (more commonly) with `delim *xy*' between .EQ and .EN. The left and right delimiters may be identical. Delimiters are turned off by `delim off'. All text that is neither between delimiters nor between .EQ and .EN is passed through untouched.

The program *checkeq* reports missing or unbalanced delimiters and .EQ/.EN pairs.

Tokens within *eqn* are separated by spaces, tabs, newlines, braces, double quotes, tildes or circumflexes. Braces {} are used for grouping; generally speaking, anywhere a single character like *x* could appear, a complicated construction enclosed in braces may be used instead. Tilde ~ represents a full space in the output, circumflex ^ half as much.

SEE ALSO

troff(1), tbl(1), ms(7), eqnchar(7)
B. W. Kernighan and L. L. Cherry, *Typesetting Mathematics-User's Guide*
J. F. Ossanna, *NROFF/TROFF User's Manual*

NOTES

To embolden digits, parens, etc., it is necessary to quote them, as in `bold "12.3"'.

NAME

ex - text editor

SYNTAX

ex [-] [-v] [-t tag] [-r] [+*lineno*] name ...

DESCRIPTION

Ex is the root of a family of editors: *edit*, *ex* and *vi*. *Ex* is a superset of *ed*, with the most notable extension being a display editing facility. Display based editing is the focus of *vi*.

If you have not used *ed*, or are a casual user, you will find that the editor *edit* is convenient for you. It avoids some of the complexities of *ex* used mostly by systems programmers and persons very familiar with *ed*.

If you have a CRT terminal, you may wish to use a display based editor; in this case see *vi*(UCB), which is a command which focuses on the display editing portion of *ex*.

DOCUMENTATION

For *edit* and *ex* see the *Ex/edit command summary - Version 2.0*. The document *Edit: A tutorial* provides a comprehensive introduction to *edit* assuming no previous knowledge of computers or the UNIX system.

The *Ex Reference Manual - Version 2.0* is a comprehensive and complete manual for the command mode features of *ex*, but you cannot learn to use the editor by reading it. For an introduction to more advanced forms of editing using the command mode of *ex* see the editing documents written by Brian Kernighan for the editor *ed*; the material in the introductory and advanced documents works also with *ex*.

An Introduction to Display Editing with Vi introduces the display editor *vi* and provides reference material on *vi*. The *Vi Quick Reference* card summarizes the commands of *vi* in a useful, functional way, and is useful with the *Introduction*.

FOR ED USERS

If you have used *ed* you will find that *ex* has a number of new features useful on CRT terminals. Intelligent terminals and high speed terminals are very pleasant to use with *vi*. Generally, the editor uses far more of the capabilities of terminals than *ed* does, and uses the terminal capability data base *termcap*(UCB) and the type of the terminal you are using from the variable TERM in the environment to determine how to drive your terminal efficiently. The editor makes use of features such as insert and delete character and line in its **visual** command (which can be abbreviated **vi**) and which is the central mode of editing when using *vi*(UCB).

There is also an interline editing **open** (**o**) command which works on all terminals.

Ex contains a number of new features for easily viewing the text of the file. The **z** command gives easy access to windows of text. Hitting ^D causes the editor to scroll a half-window of text and is more useful for quickly stepping through a file than just hitting return. Of course, the screen oriented **visual** mode gives constant access to editing context.

Ex gives you more help when you make mistakes. The **undo** (**u**) command allows you to reverse any single change which goes astray. *Ex* gives you a lot of feedback, normally printing changed lines, and indicates when more than a few lines are affected by a command so that it is easy to detect when a command has affected more lines than it should have.

The editor also normally prevents overwriting existing files unless you edited them so that you don't accidentally clobber with a *write* a file other than the one you are editing. If the system (or editor) crashes, or you accidentally hang up the phone, you can use the editor **recover** command to retrieve your work. This will get you back to within a few lines of where you left off.

Ex has several features for dealing with more than one file at a time. You can give it a list of files on the command line and use the **next** (**n**) command to deal with each in turn. The **next** command can also be given a list of file names, or a pattern as used by the shell to specify a new set of files to be dealt with. In general, filenames in the editor may be formed with full shell metasyntax. The metacharacter `%' is also available in forming filenames and is replaced by the name of the current file. For editing large groups of related files you can use *ex's* **tag** command to quickly locate functions and other important points in any of the files. This is useful when working on a large program when you want to quickly find the definition of a particular function. The command *ctags*(UCB) builds a *tags* file or a group of C programs.

For moving text between files and within a file the editor has a group of buffers, named *a* through *z*. You can place text in these named buffers and carry it over when you edit another file.

There is a command & in *ex* which repeats the last **substitute** command. In addition there is a confirmed substitute command. You give a range of substitutions to be done and the editor interactively asks whether each substitution is desired.

You can use the **substitute** command in ex to systematically convert the case of letters between upper and lower case. It is possible to ignore case of letters in searches and substitutions. Ex also allows regular expressions which match words to be constructed. This is convenient, for example, in searching for the word ``edit'' if your document also contains the word ``editor.''

Ex has a set of options which you can set to tailor it to your liking. One option which is very useful is the autoindent option which allows the editor to automatically supply leading white space to align text. You can then use the ^D key as a backtab and space and tab forward to align new code easily.

Miscellaneous new useful features include an intelligent **join** (j) command which supplies white space between joined lines automatically, commands < and > which shift groups of lines, and the ability to filter portions of the buffer through commands such as sort.

FILES

/usr/lib/ex2.0strings	error messages
/usr/lib/ex2.0recover	recover command
/usr/lib/ex2.0preserve	preserve command
/etc/termcap	describes capabilities of terminals
~/.exrc	editor startup file
/tmp/Exnnnnn	editor temporary
/tmp/Rxnnnnn	named buffer temporary
/usr/preserve	preservation directory

SEE ALSO

awk(1), ed(1), grep(1), sed(1), edit(UCB), grep(UCB), termcap(UCB), vi(UCB)

AUTHOR

William Joy

NOTES

The undo command causes all marks to be lost on lines changed and then restored if the marked lines were changed.

Undo never clears the buffer modified condition.

The z command prints a number of logical rather than physical lines. More than a screen full of output may result if long lines are present.

File input/output errors don't print a name if the command line `-' option is used.

There is no easy way to do a single scan ignoring case.

Because of the implementation of the arguments to next, only 512 bytes of argument list are allowed there.

The format of /etc/termcap and the large number of capabilities of terminals used by the editor cause terminal type setup to be rather slow.

The editor does not warn if text is placed in named buffers and not used before exiting the editor.

Null characters are discarded in input files, and cannot appear in resultant files.

NAME

grep, egrep, fgrep - search a file for a pattern

SYNTAX

grep [option] ... expression [file] ...

egrep [option] ... [expression] [file] ...

fgrep [option] ... [strings] [file]

DESCRIPTION

Commands of the *grep* family search the input *files* (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output; unless the **-h** flag is used, the file name is shown if there is more than one input file.

Grep patterns are limited regular expressions in the style of *ed*(1); it uses a compact nondeterministic algorithm. *Egrep* patterns are full regular expressions; it uses a fast deterministic algorithm that sometimes needs exponential space. *Fgrep* patterns are fixed strings; it is fast and compact.

The following options are recognized.

-v All lines but those matching are printed.

-c Only a count of matching lines is printed.

-l The names of files with matching lines are listed (once) separated by newlines.

-n Each line is preceded by its line number in the file.

-b Each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.

-s No output is produced, only status.

-h Do not print filename headers with output lines.

-y Alphabetic letters in the pattern will match letters of either case in the input (*grep* and *fgrep* only).

-e *expression*
Same as a simple *expression* argument, but useful when the *expression* begins with a -.

-f *file*
The regular expression (*egrep*) or string list (*fgrep*)

is taken from the _file_.

-x (Exact) only lines matched in their entirety are printed (_fgrep_ only).

Care should be taken when using the characters $ * [^ | ? ' " ( ) and \ in the _expression_ as they are also meaningful to the Shell. It is safest to enclose the entire _expression_ argument in single quotes ' '.

Fgrep searches for lines that contain one of the (newline-separated) _strings_.

Egrep accepts extended regular expressions. In the following description `character' excludes newline:

A \ followed by a single character matches that character.

The character ^ ($) matches the beginning (end) of a line.

A . matches any character.

A single character not otherwise endowed with special meaning matches that character.

A string enclosed in brackets [] matches any single character from the string. Ranges of ASCII character codes may be abbreviated as in `a-z0-9'. A] may occur only as the first character of the string. A literal - must be placed where it can't be mistaken as a range indicator.

A regular expression followed by * (+, ?) matches a sequence of 0 or more (1 or more, 0 or 1) matches of the regular expression.

Two regular expressions concatenated match a match of the first followed by a match of the second.

Two regular expressions separated by | or newline match either a match for the first or a match for the second.

A regular expression enclosed in parentheses matches a match for the regular expression.

The order of precedence of operators at the same parenthesis level is [] then *+? then concatenation then | and newline.

SEE ALSO

ed(1), sed(1), sh(1)

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files.

NOTES

Ideally there should be only one *grep*, but we don't know a single algorithm that spans a wide enough range of space-time tradeoffs.

Lines are limited to 256 characters; longer lines are truncated.

NAME

head - give first few lines of a stream

SYNTAX

head [-count] [file ...]

DESCRIPTION

This filter gives the first *count* lines of each of the specified files, or of the standard input. If *count* is omitted it defaults to 10.

SEE ALSO

tail(1)

AUTHOR

Bill Joy

NAME

prep - prepare text for statistical processing

SYNTAX

prep [**-dio**] file ...

DESCRIPTION

Prep reads each *file* in sequence and writes it on the standard output, one `word' to a line. A word is a string of alphabetic characters and imbedded apostrophes, delimited by space or punctuation. Hyphented words are broken apart; hyphens at the end of lines are removed and the hyphenated parts are joined. Strings of digits are discarded.

The following option letters may appear in any order:

-d Print the word number (in the input stream) with each word.

-i Take the next *file* as an `ignore' file. These words will not appear in the output. (They will be counted, for purposes of the **-d** count.)

-o Take the next *file* as an `only' file. Only these words will appear in the output. (All other words will also be counted for the **-d** count.)

-p Include punctuation marks (single nonalphanumeric characters) as separate output lines. The punctuation marks are not counted for the **-d** count.

Ignore and only files contain words, one per line.

SEE ALSO

deroff(1)

NAME

ptx - permuted index

SYNTAX

ptx [option] ... [input [output]]

DESCRIPTION

Ptx generates a permuted index to file *input* on file *output* (standard input and output default). It has three phases: the first does the permutation, generating one line for each keyword in an input line. The keyword is rotated to the front. The permuted file is then sorted. Finally, the sorted lines are rotated so the keyword comes at the middle of the page. *Ptx* produces output in the form:

.xx "tail" "before keyword" "keyword and after" "head"

where .xx may be an *nroff* or *troff*(1) macro for user-defined formatting. The *before keyword* and *keyword and after* fields incorporate as much of the line as will fit around the keyword when it is printed at the middle of the page. *Tail* and *head*, at least one of which is an empty string "", are wrapped-around pieces small enough to fit in the unused space at the opposite end of the line. When original text must be discarded, `/' marks the spot.

The following options can be applied:

-f Fold upper and lower case letters for sorting.

-t Prepare the output for the phototypesetter; the default line length is 100 characters.

-w *n* Use the next argument, *n*, as the width of the output line. The default line length is 72 characters.

-g *n* Use the next argument, *n*, as the number of characters to allow for each gap among the four parts of the line as finally printed. The default gap is 3 characters.

-o only
Use as keywords only the words given in the *only* file.

-i ignore
Do not use as keywords any words given in the *ignore* file. If the -*i* and -*o* options are missing, use /usr/lib/eign as the *ignore* file.

-b break
Use the characters in the *break* file to separate words. In any case, tab, newline, and space characters are always used as break characters.

-r Take any leading nonblank characters of each input line to be a reference identifier (as to a page or chapter) separate from the text of the line. Attach that identifier as a 5th field on each output line.

The index for this manual was generated using ptx.

FILES

/bin/sort
/usr/lib/eign

NOTES

Line length counts do not account for overstriking or proportional spacing.

NAME

pubindex - make inverted bibliographic index

SYNTAX

pubindex [file] ...

DESCRIPTION

Pubindex makes a hashed inverted index to the named *files* for use by *refer*(1). The *files* contain bibliographic references separated by blank lines. A bibliographic reference is a set of lines that contain bibliographic information fields. Each field starts on a line beginning with a `%', followed by a key-letter, followed by a blank, and followed by the contents of the field, which continues until the next line starting with `%'. The most common key-letters and the corresponding fields are:

```
A    Author name
B    Title of book containing article referenced
C    City
D    Date
d    Alternate date
E    Editor of book containing article referenced
G    Government (CFSTI) order number
I    Issuer (publisher)
J    Journal
K    Other keywords to use in locating reference
M    Technical memorandum number
N    Issue number within volume
O    Other commentary to be printed at end of reference
P    Page numbers
R    Report number
r    Alternate report number
T    Title of article, book, etc.
V    Volume number
X    Commentary unused by pubindex
```

Except for `A', each field should only be given once. Only relevant fields should be supplied. An example is:

```
%T 5-by-5 Palindromic Word Squares
%A M. D. McIlroy
%J Word Ways
%V 9
%P 199-202
%D 1976
```

FILES

x.ia, *x.ib*, *x.ic* where *x* is the first argument.

SEE ALSO

refer(1)

NAME

refer, lookbib - find and insert literature references in documents

SYNTAX

refer [option] ...

lookbib [file] ...

DESCRIPTION

Lookbib accepts keywords from the standard input and searches a bibliographic data base for references that contain those keywords anywhere in title, author, journal name, etc. Matching references are printed on the standard output. Blank lines are taken as delimiters between queries.

Refer is a preprocessor for *nroff* or *troff*(1) that finds and formats references. The input files (standard input default) are copied to the standard output, except for lines between .[and .] command lines, which are assumed to contain keywords as for *lookbib*, and are replaced by information from the bibliographic data base. The user may avoid the search, override fields from it, or add new fields. The reference data, from whatever source, are assigned to a set of *troff* strings. Macro packages such as *ms*(7) print the finished reference text from these strings. A flag is placed in the text at the point of reference; by default the references are indicated by numbers.

The following options are available:

-a*r* Reverse the first *r* author names (Jones, J. A. instead of J. A. Jones). If *r* is omitted all author names are reversed.

-b Bare mode: do not put any flags in text (neither numbers nor labels).

-c*string*
Capitalize (with CAPS SMALL CAPS) the fields whose key-letters are in *string*.

-e Instead of leaving the references where encountered, accumulate them until a sequence of the form

```
.[
$LIST$
.]
```

is encountered, and then write out all references collected so far. Collapse references to the same source.

-k*x* Instead of numbering references, use labels as

specified in a reference data line beginning %*x*; by default *x* is **L.**

-l*m*,*n* Instead of numbering references, use labels made from the senior author's last name and the year of publication. Only the first *m* letters of the last name and the last *n* digits of the date are used. If either *m* or ,*n* is omitted the entire name or date respectively is used.

-p Take the next argument as a file of references to be searched. The default file is searched last.

-n Do not search the default file.

-s*keys*
Sort references by fields whose key-letters are in the *keys* string; permute reference numbers in text accordingly. Implies **-e**. The key-letters in *keys* may be followed by a number to indicate how many such fields are used, with **+** taken as a very large number. The default is **AD** which sorts on the senior author and then date; to sort, for example, on all authors and then title use **-sA+T**.

To use your own references, put them in the format described in *pubindex*(1) They can be searched more rapidly by running *pubindex*(1) on them before using *refer*; failure to index results in a linear search.

When *refer* is used with *eqn*, *neqn* or *tbl*, *refer* should be first, to minimize the volume of data passed through pipes.

FILES

/*usr*/*dict*/*papers* directory of default publication lists and indexes
/*usr*/*lib*/*refer* directory of programs

SEE ALSO

troff(1)

NAME

rev - reverse lines of a file

SYNTAX

rev [file] ...

DESCRIPTION

Rev copies the named files to the standard output, reversing the order of characters in every line. If no file is specified, the standard input is copied.

NAME

roff - format text

SYNTAX

roff [**+n**] [**-n**] [**-s**] [**-h**] file ...

nroff -mr [option] ... file ...
troff -mr [option] ... file ...

DESCRIPTION

Roff formats text according to control lines embedded in the text in the given files. Encountering a nonexistent file terminates printing. Incoming inter-terminal messages are turned off during printing. The optional flag arguments mean:

+n Start printing at the first page with number n.
-n Stop printing at the first page numbered higher than n.
-s Stop before each page (including the first) to allow paper manipulation; resume on receipt of an interrupt signal.
-h Insert tabs in the output stream to replace spaces whenever appropriate.

Input consists of intermixed text lines, which contain information to be formatted, and request lines, which contain instructions about how to format it. Request lines begin with a distinguished control character, normally a period.

Output lines may be filled as nearly as possible with words without regard to input lineation. Line breaks may be caused at specified places by certain commands, or by the appearance of an empty input line or an input line beginning with a space.

The capabilities of roff are specified in the attached Request Summary. Numerical values are denoted there by n or +n, titles by t, and single characters by c. Numbers denoted +n may be signed + or -, in which case they signify relative changes to a quantity, otherwise they signify an absolute resetting. Missing n fields are ordinarily taken to be 1, missing t fields to be empty, and c fields to shut off the appropriate special interpretation.

Running titles usually appear at top and bottom of every page. They are set by requests like

```
.he 'part1'part2'part3'
```

Part1 is left justified, part2 is centered, and part3 is right justified on the page. Any % sign in a title is replaced by the current page number. Any nonblank may serve

as a quote.

ASCII tab characters are replaced in the input by a _replacement character_, normally a space, according to the column settings given by a .ta command. (See .tc for how to convert this character on output.)

Automatic hyphenation of filled output is done under control of .hy. When a word contains a designated _hyphenation character_, that character disappears from the output and hyphens can be introduced into the word at the marked places only.

The **-mr** option of _nroff_ or _troff_(1) simulates _roff_ to the greatest extent possible.

FILES

/usr/lib/suftab suffix hyphenation tables
/tmp/rtm? temporary

NOTES

Roff is the simplest of the text formatting programs, and is utterly frozen.

REQUEST SUMMARY

Request	Break	Initial	Meaning
.ad	yes	yes	Begin adjusting right margins.
.ar	no	arabic	Arabic page numbers.
.br	yes	-	Causes a line break the filling of the current line is stopped.
.bl n	yes	-	Insert of n blank lines, on new page if necessary.
.bp +n	yes	n=1	Begin new page and number it n; no n means `+1'.
.cc c	no	c=.	Control character becomes `c'.
.ce n	yes	-	Center the next n input lines, without filling.
.de xx	no	-	Define parameterless macro to be invoked by request `.xx' (definition ends on line beginning `..').
.ds	yes	no	Double space; same as `.ls 2'.
.ef t	no	t=	Even foot title becomes t.
.eh t	no	t=	Even head title becomes t.
.fi	yes	yes	Begin filling output lines.
.fo	no	t=	All foot titles are t.
.hc c	no	none	Hyphenation character becomes `c'.
.he t	no	t=	All head titles are t.
.hx	no	-	Title lines are suppressed.
.hy n	no	n=1	Hyphenation is done, if n=1; and is not done, if n=0.
.ig	no	-	Ignore input lines through a line beginning with `..'.
.in +n	yes	-	Indent n spaces from left margin.
.ix +n	no	-	Same as `.in' but without break.
.li n	no	-	Literal, treat next n lines as text.
.ll +n	no	n=65	Line length including indent is n characters.
.ls +n	yes	n=1	Line spacing set to n lines per output line.
.ml n	no	n=2	Put n blank lines between the top of page and head title.
.m2 n	no	n=2	n blank lines put between head title and beginning of text on page.
.m3 n	no	n=1	n blank lines put between end of text and foot title.
.m4 n	no	n=3	n blank lines put between the foot title and the bottom of page.
.na	yes	no	Stop adjusting the right margin.
.ne n	no	-	Begin new page, if n output lines cannot fit on present page.
.nn +n	no	-	The next n output lines are not numbered.
.nl	no	no	Add 5 to page offset; number lines in margin from 1 on each page.
.n2 n	no	no	Add 5 to page offset; number lines from n; stop if n=0.
.ni +n	no	n=0	Line numbers are indented n.
.nf	yes	no	Stop filling output lines.

```
.nx file          -    Switch input to `file'.
.of t      no     t=   Odd foot title becomes t.
.oh t      no     t=   Odd head title becomes t.
.pa +n     yes    n=1  Same as `.bp'.
.pl +n     no     n=66 Total paper length taken to be n lines.
.po +n     no     n=0  Page offset.  All lines are preceded by n
                  spaces.
.ro  no    arabic      Roman page numbers.
.sk n      no     -    Produce n blank pages starting next page.
.sp n      yes    -    Insert block of n blank lines, except at top
                  of page.
.ss  yes   yes    Single space output lines, equivalent to `.ls 1'.
.ta n n..         -    Pseudotab settings.
                  .ta n n..      -    Pseudotab settings.  Initial
                  tab settings are columns 9 17 25 ...
.tc c      no     space     Tab replacement character becomes `c'.
.ti +n     yes    -    Temporarily indent next output line n spaces.
.tr cdef..        no   -    Translate c into d, e into f, etc.
.ul n      no     -    Underline the letters and numbers in the next
                  n input lines.
```

NAME

sed - stream editor

SYNTAX

sed [**-n**] [**-e** script] [**-f** sfile] [file] ...

DESCRIPTION

Sed copies the named files (standard input default) to the standard output, edited according to a script of commands. The **-f** option causes the script to be taken from file sfile; these options accumulate. If there is just one **-e** option and no **-f**'s, the flag **-e** may be omitted. The **-n** option suppresses the default output.

A script consists of editing commands, one per line, of the following form:

[address [, address]] function [arguments]

In normal operation sed cyclically copies a line of input into a pattern space (unless there is something left after a `D' command), applies in sequence all commands whose addresses select that pattern space, and at the end of the script copies the pattern space to the standard output (except under **-n**) and deletes the pattern space.

An address is either a decimal number that counts input lines cumulatively across files, a `$' that addresses the last line of input, or a context address, `/regular expression/', in the style of ed(1) modified thus:

The escape sequence `\n' matches a newline embedded in the pattern space.

A command line with no addresses selects every pattern space.

A command line with one address selects each pattern space that matches the address.

A command line with two addresses selects the inclusive range from the first pattern space that matches the first address through the next pattern space that matches the second. (If the second address is a number less than or equal to the line number first selected, only one line is selected.) Thereafter the process is repeated, looking again for the first address.

Editing commands can be applied only to non-selected pattern spaces by use of the negation function `!' (below).

In the following list of functions the maximum number of permissible addresses for each function is indicated in parentheses.

An argument denoted text consists of one or more lines, all but the last of which end with `\' to hide the newline. Backslashes in text are treated like backslashes in the replacement string of an `s' command, and may be used to protect initial blanks and tabs against the stripping that is done on every script line.

An argument denoted rfile or wfile must terminate the command line and must be preceded by exactly one blank. Each wfile is created before processing begins. There can be at most 10 distinct wfile arguments.

(1)a\
text
: Append. Place text on the output before reading the next input line.

(2)b label
: Branch to the `:' command bearing the label. If label is empty, branch to the end of the script.

(2)c\
text
: Change. Delete the pattern space. With 0 or 1 address or at the end of a 2-address range, place text on the output. Start the next cycle.

(2)d Delete the pattern space. Start the next cycle.

(2)D Delete the initial segment of the pattern space through the first newline. Start the next cycle.

(2)g Replace the contents of the pattern space by the contents of the hold space.

(2)G Append the contents of the hold space to the pattern space.

(2)h Replace the contents of the hold space by the contents of the pattern space.

(2)H Append the contents of the pattern space to the hold space.

(1)i\
text Insert. Place text on the standard output.

(2)l List the pattern space on the standard output in an

unambiguous form. Non-printing characters are spelled in two digit ascii, and long lines are folded.

(2)n Copy the pattern space to the standard output. Replace the pattern space with the next line of input.

(2)N Append the next line of input to the pattern space with an embedded newline. (The current line number changes.)

(2)p Print. Copy the pattern space to the standard output.

(2)P Copy the initial segment of the pattern space through the first newline to the standard output.

(1)q Quit. Branch to the end of the script. Do not start a new cycle.

(2)r _rfile_
Read the contents of _rfile_. Place them on the output before reading the next input line.

(2)s/_regular expression_/_replacement_/_flags_
Substitute the _replacement_ string for instances of the _regular expression_ in the pattern space. Any character may be used instead of \`/'. For a fuller description see _ed_(1). _Flags_ is zero or more of

g Global. Substitute for all nonoverlapping instances of the _regular expression_ rather than just the first one.

p Print the pattern space if a replacement was made.

w _wfile_
Write. Append the pattern space to _wfile_ if a replacement was made.

(2)t _label_
Test. Branch to the \`:' command bearing the _label_ if any substitutions have been made since the most recent reading of an input line or execution of a \`t'. If _label_ is empty, branch to the end of the script.

(2)w _wfile_
Write. Append the pattern space to _wfile_.

(2)x Exchange the contents of the pattern and hold spaces.

(2)y/_string1_/_string2_/
Transform. Replace all occurrences of characters in _string1_ with the corresponding character in _string2_.

The lengths of _string1_ and _string2_ must be equal.

(2)! _function_
Don't. Apply the _function_ (or group, if _function_ is `{') only to lines _not_ selected by the address(es).

(0): _label_
This command does nothing; it bears a _label_ for `b' and `t' commands to branch to.

(1)= Place the current line number on the standard output as a line.

(2){ Execute the following commands through a matching `}' only when the pattern space is selected.

(0) An empty command is ignored.

SEE ALSO

ed(1), grep(1), awk(1)

NAME

sort - sort or merge files

SYNTAX

sort [-*mubdfinrtx*] [+*pos1* [-*pos2*]] ... [**-o** name] [-T directory] [name] ...

DESCRIPTION

Sort sorts lines of all the named files together and writes the result on the standard output. The name '-' means the standard input. If no input files are named, the standard input is sorted.

The default sort key is an entire line. Default ordering is lexicographic by bytes in machine collating sequence. The ordering is affected globally by the following options, one or more of which may appear.

- **b** Ignore leading blanks (spaces and tabs) in field comparisons.
- **d** 'Dictionary' order: only letters, digits and blanks are significant in comparisons.
- **f** Fold upper case letters onto lower case.
- **i** Ignore characters outside the ASCII range 040-0176 in nonnumeric comparisons.
- **n** An initial numeric string, consisting of optional blanks, optional minus sign, and zero or more digits with optional decimal point, is sorted by arithmetic value. Option **n** implies option **b**.
- **r** Reverse the sense of comparisons.
- **t*x*** 'Tab character' separating fields is *x*.

The notation +*pos1* -*pos2* restricts a sort key to a field beginning at *pos1* and ending just before *pos2*. *Pos1* and *pos2* each have the form *m.n*, optionally followed by one or more of the flags **bdfinr**, where *m* tells a number of fields to skip from the beginning of the line and *n* tells a number of characters to skip further. If any flags are present they override all the global ordering options for this key. If the **b** option is in effect *n* is counted from the first nonblank in the field; **b** is attached independently to *pos2*. A missing .*n* means .0; a missing -*pos2* means the end of the line. Under the **-t*x*** option, fields are strings separated by *x*; otherwise fields are nonempty nonblank strings separated by blanks.

When there are multiple sort keys, later keys are compared only after all earlier keys compare equal. Lines that otherwise compare equal are ordered with all bytes significant.

These option arguments are also understood:

c Check that the input file is sorted according to the ordering rules; give no output unless the file is out of sort.

m Merge only, the input files are already sorted.

o The next argument is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs.

T The next argument is the name of a directory in which temporary files should be made.

u Suppress all but one in each set of equal lines. Ignored bytes and bytes outside keys do not participate in this comparison.

Examples. Print in alphabetical order all the unique spellings in a list of words. Capitalized words differ from uncapitalized.

```
sort -u +0f +0 list
```

Print the password file (passwd(5)) sorted by user id number (the 3rd colon-separated field).

```
sort -t: +2n /etc/passwd
```

Print the first instance of each month in an already sorted file of (month day) entries. The options **-um** with just one input file make the choice of a unique representative from a set of equal lines predictable.

```
sort -um +0 -1 dates
```

FILES

/usr/tmp/stm*, /tmp/*: first and second tries for temporary files

SEE ALSO

uniq(1), comm(1), rev(1), join(1)

DIAGNOSTICS

Comments and exits with nonzero status for various trouble conditions and for disorder discovered under option **-c**.

NOTES

Very long lines are silently truncated.

NAME

spell, spellin, spellout - find spelling errors

SYNTAX

spell [option] ... [file] ...

/usr/src/cmd/spell/spellin [list]

/usr/src/cmd/spell/spellout [**-d**] list

DESCRIPTION

Spell collects words from the named documents, and looks them up in a spelling list. Words that neither occur among nor are derivable (by applying certain inflections, prefixes or suffixes) from words in the spelling list are printed on the standard output. If no files are named, words are collected from the standard input.

Spell ignores most *troff*, *tbl* and *eqn*(1) constructions.

Under the **-v** option, all words not literally in the spelling list are printed, and plausible derivations from spelling list words are indicated.

Under the -b option, British spelling is checked. Besides preferring *centre*, *colour*, *speciality*, *travelled*, etc., this option insists upon -*ise* in words like *standardise*, Fowler and the OED to the contrary notwithstanding.

Under the **-x** option, every plausible stem is printed with \`=' for each word.

The spelling list is based on many sources, and while more haphazard than an ordinary dictionary, is also more effective in respect to proper names and popular technical words. Coverage of the specialized vocabularies of biology, medicine and chemistry is light.

Pertinent auxiliary files may be specified by name arguments, indicated below with their default settings. Copies of all output are accumulated in the history file. The stop list filters out misspellings (e.g. thier=thy-y+ier) that would otherwise pass.

Two routines help maintain the hash lists used by *spell*. Both expect a list of words, one per line, from the standard input. *Spellin* adds the words on the standard input to the preexisting *list* and places a new list on the standard output. If no *list* is specified, the new list is created from scratch. *Spellout* looks up each word in the standard input and prints on the standard output those that are missing from (or present on, with option -d) the hash list.

FILES

D=/usr/dict/hlist[ab]: hashed spelling lists, American & British
S=/usr/dict/hstop: hashed stop list
H=/usr/dict/spellhist: history file
/usr/lib/spell
deroff(1), sort(1), tee(1), sed(1)

NOTES

The spelling list's coverage is uneven; new installations will probably wish to monitor the output for several months to gather local additions.
British spelling was done by an American.

NAME

split - split a file into pieces

SYNTAX

split [-_n_] [file [name]]

DESCRIPTION

Split reads _file_ and writes it in _n_-line pieces (default 1000), as many as necessary, onto a set of output files. The name of the first output file is _name_ with **aa** appended, and so on lexicographically. If no output name is given, **x** is default.

If no input file is given, or if - is given in its stead, then the standard input file is used.

WARNING

1000 lines is usually less than 19 pages.
Lpr does not guarantee that it prints the files in the order given.

SEE ALSO

lpr (1), wc (1)

NAME

tail - deliver the last part of a file

SYNTAX

tail ±number[lbc] [file]

DESCRIPTION

Tail copies the named file to the standard output beginning at a designated place. If no file is named, the standard input is used.

Copying begins at distance +number from the beginning, or -number from the end of the input. Number is counted in units of lines, blocks or characters, according to the appended option **l**, **b** or **c**. When no units are specified, counting is by lines.

SEE ALSO

dd(1)

NOTES

Tails relative to the end of the file are treasured up in a buffer, and thus are limited in length. Various kinds of anomalous behavior may happen with character special files.

NAME

tbl - format tables for nroff or troff

SYNTAX

tbl [files] ...

DESCRIPTION

Tbl is a preprocessor for formatting tables for nroff or troff(1). The input files are copied to the standard output, except for lines between .TS and .TE command lines, which are assumed to describe tables and reformatted. Details are given in the reference manual.

As an example, letting \t represent a tab (which should be typed as a genuine tab) the input

```
.TS
c s s
c c s
c c c
l n n.
Household Population
Town\tHouseholds
\tNumber\tSize
Bedminster\t789\t3.26
Bernards Twp.\t3087\t3.74
Bernardsville\t2018\t3.30
Bound Brook\t3425\t3.04
Branchburg\t1644\t3.49
Bridgewater\t7897\t3.81
Far Hills\t240\t3.19
.TE
```

yields

```
      Household Population
      Town          Households
                    Number   Size
Bedminster            789    3.26
Bernards Twp.        3087    3.74
Bernardsville        2018    3.30
Bound Brook          3425    3.04
Branchburg           1644    3.49
Bridgewater          7897    3.81
Far Hills             240    3.19
```

If no arguments are given, tbl reads the standard input, so it may be used as a filter. When it is used with eqn or neqn the tbl command should be first, to minimize the volume of data passed through pipes.

SEE ALSO

troff(1), eqn(1)
M. E. Lesk, TBL.

NAME

tr - translate characters

SYNTAX

tr [**-cds**] [string1 [string2]]

DESCRIPTION

Tr copies the standard input to the standard output with substitution or deletion of selected characters. Input characters found in *string1* are mapped into the corresponding characters of *string2*. When *string2* is short it is padded to the length of *string1* by duplicating its last character. Any combination of the options **-cds** may be used: **-c** complements the set of characters in *string1* with respect to the universe of characters whose ASCII codes are 01 through 0377 octal; **-d** deletes all input characters in *string1*; **-s** squeezes all strings of repeated output characters that are in *string2* to single characters.

In either string the notation *a-b* means a range of characters from *a* to *b* in increasing ASCII order. The character `\' followed by 1, 2 or 3 octal digits stands for the character whose ASCII code is given by those digits. A `\' followed by any other character stands for that character.

The following example creates a list of all the words in `file1' one per line in `file2', where a word is taken to be a maximal string of alphabetics. The second string is quoted to protect `\' from the Shell. 012 is the ASCII code for newline.

```
tr -cs A-Za-z '\012' <file1 >file2
```

SEE ALSO

ed(1), ascii(7)

NOTES

Won't handle ASCII NUL in *string1* or *string2*; always deletes NUL from input.

NAME

troff, nroff - text formatting and typesetting

SYNTAX

troff [option] ... [file] ...

nroff [option] ... [file] ...

DESCRIPTION

Troff formats text in the named *files* for printing on a Graphic Systems C/A/T phototypesetter; *nroff* for typewriter-like devices. Their capabilities are described in the *Nroff/Troff user's manual*.

If no *file* argument is present, the standard input is read. An argument consisting of a single minus (-) is taken to be a file name corresponding to the standard input. The options, which may appear in any order so long as they appear before the files, are:

-o*list* Print only pages whose page numbers appear in the comma-separated *list* of numbers and ranges. A range *N*-*M* means pages *N* through *M*; an initial -*N* means from the beginning to page *N*; and a final *N*- means from *N* to the end.

-n*N* Number first generated page *N*.

-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. *Troff* will stop the phototypesetter every *N* pages, produce a trailer to allow changing cassettes, and resume when the typesetter's start button is pressed.

-m*name* Prepend the macro file **/usr/lib/tmac/tmac.***name* to the input *files*.

-r*aN* Set register *a* (one-character) to *N*.

-i Read standard input after the input files are exhausted.

-q Invoke the simultaneous input-output mode of the **rd** request.

Nroff only

-T*name* Prepare output for specified terminal. Known *names* are **37** for the (default) Teletype Corporation Model 37 terminal, **tn300** for the GE TermiNet 300 (or any terminal without half-line capability), **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.

-h Use output tabs during horizontal spacing to speed output and reduce output character count. Tab settings are assumed to be every 8 nominal character widths.

Troff only

-t Direct output to the standard output instead of the phototypesetter.

-f Refrain from feeding out paper and stopping phototypesetter at the end of the run.

-w Wait until phototypesetter is available, if currently busy.

-b Report whether the phototypesetter is busy or available. No text processing is done.

-a Send a printable ASCII approximation of the results to the standard output.

-pN Print all characters in point size *N* while retaining all prescribed spacings and motions, to reduce phototypesetter elasped time.

-g Prepare output for a GCOS phototypesetter and direct it to the standard output (see *gcat*(1)).

If the file /*usr*/*adm*/*tracct* is writable, *troff* keeps phototypesetter accounting records there. The integrity of that file may be secured by making *troff* a `set user-id' program.

FILES

/usr/lib/suftab	suffix hyphenation tables
/tmp/ta*	temporary file
/usr/lib/tmac/tmac.*	standard macro files
/usr/lib/term/*	terminal driving tables for *nroff*
/usr/lib/font/*	font width tables for *troff*
/dev/cat	phototypesetter
/usr/adm/tracct	accounting statistics for /dev/cat

SEE ALSO

J. F. Ossanna, *Nroff/Troff user's manual*
B. W. Kernighan, *A TROFF Tutorial*
eqn(1), tbl(1)

col(1), tk(1) (*nroff* only)
tc(1), gcat(1) (*troff* only)

NAME

uniq - report repeated lines in a file

SYNTAX

uniq [-udc [+n] [-n]] [input [output]]

DESCRIPTION

Uniq reads the input file comparing adjacent lines. In the normal case, the second and succeeding copies of repeated lines are removed; the remainder is written on the output file. Note that repeated lines must be adjacent in order to be found; see *sort*(1). If the **-u** flag is used, just the lines that are not repeated in the original file are output. The **-d** option specifies that one copy of just the repeated lines is to be written. The normal mode output is the union of the **-u** and **-d** mode outputs.

The **-c** option supersedes **-u** and **-d** and generates an output report in default style but with each line preceded by a count of the number of times it occurred.

The *n* arguments specify skipping an initial portion of each line in the comparison:

-*n* The first *n* fields together with any blanks before each are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbors.

+*n* The first *n* characters are ignored. Fields are skipped before characters.

SEE ALSO

sort(1), comm(1)

NAME

vi - screen oriented (visual) display editor based on ex

SYNTAX

vi [-t tag] [-r] [+*lineno*] name ...

DESCRIPTION

Vi (visual) is a display oriented text editor based on *ex*(UCB). *Ex* and *vi* run the same code; it is possible to get to the command mode of *ex* from within *vi* and vice-versa.

The *Vi Quick Reference* card and the *Introduction to Display Editing with Vi* provide full details on using *vi*.

FILES

See *ex*(UCB).

SEE ALSO

ex (UCB), vi (UCB), ``Vi Quick Reference'' card, ``An Introduction to Display Editing with Vi''.

NOTES

Scans with / and ? begin on the next line, skipping the remainder of the current line.

Software tabs using **^T** work only immediately after the *autoindent*.

Left and right shifts on intelligent terminals don't make use of insert and delete character operations in the terminal.

The *wrapmargin* option can be fooled since it looks at output columns when blanks are typed. If a long word passes through the margin and onto the next line without a break, then the line won't be broken.

Insert/delete within a line can be slow if tabs are present on intelligent terminals, since the terminals need help in doing this correctly.

Occasionally inverse video scrolls up into the file from a diagnostic on the last line.

Saving text on deletes in the named buffers is somewhat inefficient.

The *source* command does not work when executed as **:source**; there is no way to use the **:append**, **:change**, and **:insert** commands, since it is not possible to give more than one line of input to a : escape. To use these on a **:global** you must **Q** to *ex* command mode, execute them, and then reenter

the screen editor with _vi_ or _open_.

NAME

wc - word count

SYNTAX

wc [**-lwc**] [name ...]

DESCRIPTION

Wc counts lines, words and characters in the named files, or in the standard input if no name appears. A word is a maximal string of characters delimited by spaces, tabs or newlines.

If the optional argument is present, just the specified counts (lines, words or characters) are selected by the letters **l**, **w**, or **c**.