PROGRAMMER'S GUIDE
CTIX™ SUPPLEMENT

First Edition (June 1987) 09-01040-01

This manual was prepared on a Convergent Technologies S/320 Computer System and printed on an Imagen 8/300 Laser Printer.

Contents

Preface

The *Programmer's Guide: CTIX Supplement* discusses in detail various tools provided with the CTIX* operating system. This manual should be used in conjunction with the *CTIX Operating System Manual*, which is a more general and comprehensive reference on the CTIX operating system. The *Programmer's Guide: UNIX System V* should be used as a reference on tools such as *awk*, *make*, the C programming language, Common Object File Format, and so forth.

The tools described in this manual address four areas:

- **Support Tools:** the *m4* macro processor; and the calculator programs *bc* and *dc*.
- **Text Editing:** display editing with *vi*; and *sed*, a non-interactive text editor.
- **Text Formatting:** an *nroff/troff* user's guide and tutorial; preparing documents with the macro packages *-me*, *-mm* and *-me*; typesetting mathematics with *eqn*; and *tbl*, a program to format tables.
- **CTIX Programming:** Convergent Technologies S/1280 Inter-CPU communication facility; and the CTIX system assembler user's guide.

The first three parts of this manual (Chapters 1 to 13) consist of standard documentation released by AT&T that describes the UNIX Operating System or documentation originating from the University

* A trademark of Convergent Technologies, Inc. The CTIX operating system is derived from UNIX System V by Convergent Technologies under license from AT&T. UNIX is a trademark of AT&T Bell Laboratories.

of California, Berkeley. Due to the "classical" nature of these documents, their contents have not been altered with the exception of minor formatting changes needed to conform to CTIX manual style. In order to keep as true to the original documents as possible, the conventions vary from chapter to chapter; in addition, because this manual is published in a reduced format, point sizes mentioned in articles are not true to scale. Many of the articles refer to other operating systems, such as GCOS, and other hardware, such as PDP-11. In all cases, however, the documentation is compatible with the CTIX operating system.

1

The M4 Macro Processor

Abstract

M4 is a macro processor available on UNIX and GCOS. Its primary use has been as a front end for Ratfor for those cases where parameterless macros are not adequately powerful. It has also been used for languages as disparate as C and Cobol. M4 is particularly suited for functional languages like Fortran, PL/I and C since macros are specified in a functional notation.

M4 provides features seldom found even in much larger macro processors, including

- arguments
- condition testing
- arithmetic capabilities
- string and substring functions
- file manipulation

This paper is a user's manual for M4.

Source: Brian W. Kernighan and Dennis M. Ritchie, *The M4 Macro Processor* (Murray Hill, N.J.: Bell Laboratories, 1977).

Introduction

A macro processor is a useful way to enhance a programming language, to make it more palatable or more readable, or to tailor it to a particular application. The **#define** statement in C and the analogous **define** in Ratfor are examples of the basic facility provided by any macro processor—replacement of text by other text.

The M4 macro processor is an extension of a macro processor called M3 which was written by D. M. Ritchie for the AP-3 minicomputer; M3 was in turn based on a macro processor implemented for [1]. Readers unfamiliar with the basic ideas of macro processing may wish to read some of the discussion there.

M4 is a suitable front end for Ratfor and C, and has also been used successfully with Cobol. Besides the straightforward replacement of one string of text by another, it provides macros with arguments, conditional macro expansion, arithmetic, file manipulation, and some specialized string processing functions.

The basic operation of M4 is to copy its input to its output. As the input is read, however, each alphanumeric "token" (that is, string of letters and digits) is checked. If it is the name of a macro, then the name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the defining text before it is rescanned.

M4 provides a collection of about twenty built-in macros which perform various useful operations; in addition, the user can define new macros. Built-ins and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process.

Usage

On UNIX, use

m4 [files]

Each argument file is processed in order; if there are no arguments, or if an argument is '–', the standard input is read at that point. The processed text is written on the standard output, which may be

captured for subsequent processing with

m4 [files] >outputfile

On GCOS, usage is identical, but the program is called **./m4 .**

Defining Macros

The primary built-in function of M4 is **define**, which is used to define new macros. The input

define(name, stuff)

causes the string **name** to be defined as **stuff**. All subsequent occurrences of **name** will be replaced by **stuff**. **name** must be alphanumeric and must begin with a letter (the underscore _ counts as a letter). **stuff** is any text that contains balanced parentheses; it may stretch over multiple lines.

Thus, as a typical example,

```
define(N, 100)
 ...
if (i > N)
```

defines N to be 100, and uses this "symbolic constant" in a later **if** statement.

The left parenthesis must immediately follow the word **define**, to signal that **define** has arguments. If a macro or built-in name is not followed immediately by '(', it is assumed to have no arguments. This is the situation for N above; it is actually a macro with no arguments, and thus when it is used there need be no (...) following it.

You should also notice that a macro name is only recognized as such if it appears surrounded by non-alphanumerics. For example, in

```
define(N, 100)
 ...
if (NNN > 100)
```

the variable NNN is absolutely unrelated to the defined macro N, even though it contains a lot of N's.

Things may be defined in terms of other things. For example,

```
define(N, 100)
define(M, N)
```

defines both M and N to be 100.

What happens if **N** is redefined? Or, to say it another way, is **M** defined as **N** or as 100? In M4, the latter is true—**M** is 100, so even if **N** subsequently changes, **M** does not.

This behavior arises because M4 expands macro names into their defining text as soon as it possibly can. Here, that means that when the string **N** is seen as the arguments of **define** are being collected, it is immediately replaced by 100; it's just as if you had said

```
define(M, 100)
```

in the first place.

If this isn't what you really want, there are two ways out of it. The first, which is specific to this situation, is to interchange the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now **M** is defined to be the string **N**, so when you ask for **M** later, you'll always get the value of **N** at that time (because the **M** will be replaced by **N** which will be replaced by 100).

Quoting

The more general solution is to delay the expansion of the arguments of **define** by *quoting* them. Any text surrounded by the single quotes ` and ´ is not expanded immediately, but has the quotes stripped off. If you say

```
define(N, 100)
define(M, `N')
```

the quotes around the **N** are stripped off as the argument is being collected, but they have served their purpose, and **M** is defined as the string **N**, not 100. The general rule is that M4 always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If you want the word **define** to appear in the output, you have to quote it in the input, as in

```
`define' = 1;
```

As another instance of the same thing, which is a bit more surprising, consider redefining **N**:

```
define(N, 100)
...
define(N, 200)
```

Perhaps regrettably, the **N** in the second definition is evaluated as soon as it's seen; that is, it is replaced by 100, so it's as if you had written

```
define(100, 200)
```

This statement is ignored by M4, since you can only define things that look like names, but it obviously doesn't have the effect you wanted. To really redefine **N**, you must delay the evaluation by quoting:

```
define(N, 100)
...
define('N', 200)
```

In M4, it is often wise to quote the first argument of a macro.

If ` and ' are not convenient for some reason, the quote characters can be changed with the built-in **changequote**:

```
changequote([, ])
```

makes the new quote characters the left and right brackets. You can restore the original characters with just

```
changequote
```

There are two additional built-ins related to **define**. **undefine** removes the definition of some macro or built-in:

```
undefine('N')
```

removes the definition of **N**. (Why are the quotes absolutely necessary?) Built-ins can be removed with **undefine**, as in

```
undefine('define')
```

but once you remove one, you can never get it back.

The built-in **ifdef** provides a way to determine if a macro is currently defined. In particular, M4 has pre-defined the names **unix** and **gcos** on the corresponding systems, so you can tell which one you're using:

```
ifdef('unix', 'define(wordsize,16)' )
ifdef('gcos', 'define(wordsize,36)' )
```

makes a definition appropriate for the particular machine. Don't forget the quotes!

ifdef actually permits three arguments; if the name is undefined, the value of **ifdef** is then the third argument, as in

```
ifdef('unix', on UNIX, not on UNIX)
```

Arguments

So far we have discussed the simplest form of macro processing—replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**) any occurrence of **$n** will be replaced by the **n**th argument when the macro is actually used. Thus, the macro **bump**, defined as

```
define(bump, $1 = $1 + 1)
```

generates code to increment its argument by 1:

```
bump(x)
```

is

```
x = x + 1
```

A macro can have as many arguments as you want, but only the first nine are accessible, through **$1** to **$9**. (The macro name itself is **$0**, although that is less commonly used.) Arguments that are not supplied are replaced by null strings, so we can define a macro **cat** which simply concatenates its arguments, like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus

```
cat(x, y, z)
```

is equivalent to

```
xyz
```

$4 through **$9** are null, since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus

```
define(a,   b   c)
```

defines **a** to be **b c**.

Arguments are separated by commas, but parentheses are counted properly, so a comma "protected" by parentheses does not terminate an argument. That is, in

```
define(a, (b,c))
```

there are only two arguments; the second is literally **(b,c)**. And of course a bare comma or parenthesis can be inserted by quoting it.

Arithmetic Built-ins

M4 provides two built-in functions for doing arithmetic on integers (only). The simplest is **incr**, which increments its numeric argument by 1. Thus to handle the common programming situation where you want a variable to be defined as "one more than N", write

```
define(N, 100)
define(N1, 'incr(N)')
```

Then **N1** is defined as one more than the current value of **N**.

The more general mechanism for arithmetic is a built-in called **eval**, which is capable of arbitrary arithmetic on integers. It provides the operators (in decreasing order of precedence)

```
unary + and -
** or ^              (exponentiation)
*  /  %(modulus)
+  -
==  !=  <  <=  >  >=
!              (not)
& or &&        (logical and)
| or ||        (logical or)
```

Parentheses may be used to group operations where needed. All the

operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1, and false is 0. The precision in **eval** is 32 bits on UNIX and 36 bits on GCOS.

As a simple example, suppose we want **M** to be **2**N+1**. Then

```
define(N, 3)
define(M, 'eval(2**N+1)')
```

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (say just a number); it usually gives the result you want, and is a good habit to get into.

File Manipulation

You can include a new file in the input at any time by the built-in function **include**:

```
include(filename)
```

inserts the contents of **filename** in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used; **sinclude** ("silent include") says nothing and continues if it can't access the file.

It is also possible to divert the output of M4 to temporary files during processing, and output the collected material upon command. M4 maintains nine of these diversions, numbered 1 through 9. If you say

```
divert(n)
```

all subsequent output is put onto the end of a temporary file referred to as **n**. Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

```
undivert
```

brings back all diversions in numeric order, and **undivert** with

arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of **undivert** is *not* the diverted stuff. Furthermore, the diverted material is *not* rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. This is zero during normal processing.

System Command

You can run any program in the local operating system with the **syscmd** built-in. For example,

syscmd(date)

on UNIX runs the **date** command. Normally **syscmd** would be used to create a file for a subsequent **include**.

To facilitate making unique file names, the built-in **maketemp** is provided, with specifications identical to the system function *mktemp*: a string of XXXXX in the argument is replaced by the process id of the current process.

Conditionals

There is a built-in called **ifelse** which enables you to perform arbitrary conditional testing. In the simplest form,

ifelse(a, b, c, d)

compares the two strings **a** and **b**. If these are identical, **ifelse** returns the string **c**; otherwise it returns **d**. Thus we might define a macro called **compare** which compares two strings and returns "yes" or "no" if they are the same or different.

define(compare, 'ifelse($1, $2, yes, no)')

Note the quotes, which prevent too-early evaluation of **ifelse**.

If the fourth argument is missing, it is treated as empty.

ifelse can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. In the input

ifelse(a, b, c, d, e, f, g)

if the string **a** matches the string **b**, the result is **c**. Otherwise, if **d** is the same as **e**, the result is **f**. Otherwise the result is **g**. If the final argument is omitted, the result is null, so

ifelse(a, b, c)

is **c** if **a** matches **b**, and null otherwise.

String Manipulation

The built-in **len** returns the length of the string that makes up its argument. Thus

len(abcdef)

is 6, and **len((a,b))** is 5.

The built-in **substr** can be used to produce substrings of strings. **substr(s, i, n)** returns the substring of **s** that starts at the **i**th position (origin zero), and is **n** characters long. If **n** is omitted, the rest of the string is returned, so

substr('now is the time', 1)

is

ow is the time

If **i** or **n** are out of range, various sensible things happen.

index(s1, s2) returns the index (position) in **s1** where the string **s2** occurs, or –1 if it doesn't occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration.

translit(s, f, t)

modifies **s** by replacing any character found in **f** by the corresponding character of **t**. That is,

translit(s, aeiou, 12345)

replaces the vowels by the corresponding digits. If **t** is shorter than **f**, characters which don't have an entry in **t** are deleted; as a limiting case, if **t** is not present at all, characters from **f** are deleted from **s**.

So

```
translit(s, aeiou)
```

deletes vowels from **s**.

There is also a built-in called **dnl** which deletes all characters that follow it up to and including the next newline; it is useful mainly for throwing away empty lines that otherwise tend to clutter up M4 output. For example, if you say

```
define(N, 100)
define(M, 200)
define(L, 300)
```

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add **dnl** to each of these lines, the newlines will disappear.

Another way to achieve this, due to J. E. Weythman, is

```
divert(-1)
        define(...)
        ...
divert
```

Printing

The built-in **errprint** writes its arguments out on the standard error file. Thus you can say

```
errprint('fatal error')
```

dumpdef is a debugging aid which dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise you get the ones you name as arguments. Don't forget to quote the names!

Summary of Built-ins

Each entry is preceded by the page number where it is described.

Acknowledgements

We are indebted to Rick Becker, John Chambers, Doug McIlroy, and especially Jim Weythman, whose pioneering use of M4 has led to several valuable improvements. We are also deeply grateful to Weythman for several substantial contributions to the code.

References

[1] B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, Inc., 1976.

2

BC — An Arbitrary Precision Desk-Calculator Language

Abstract

BC is a language and a compiler for doing arbitrary precision arithmetic on the PDP-11 under the UNIX time-sharing system. The output of the compiler is interpreted and executed by a collection of routines which can input, output, and do arithmetic on indefinitely large integers and on scaled fixed-point numbers.

These routines are themselves based on a dynamic storage allocator. Overflow does not occur until all available core storage is exhausted.

The language has a complete control structure as well as immediate-mode operation. Functions can be defined and saved for later execution.

Two five hundred-digit numbers can be multiplied to give a thousand digit result in about ten seconds.

A small collection of library functions is also available, including sin, cos, arctan, log, exponential, and Bessel functions of integer order.

Some of the uses of this compiler are

- to do computation with large integers,
- to do computation accurate to many decimal places,
- conversion of numbers from one base to another base.

Source: Lorinda Cherry, Robert Morris, *BC — An Arbitrary Precision Desk-Calculator Language* (Murray Hill, N.J.: Bell Laboratories, 1978).

Introduction

BC is a language and a compiler for doing arbitrary precision arithmetic on the UNIX time-sharing system [1]. The compiler was written to make conveniently available a collection of routines (called DC [5]) which are capable of doing arithmetic on integers of arbitrary size. The compiler is by no means intended to provide a complete programming language. It is a minimal language facility.

There is a scaling provision that permits the use of decimal point notation. Provision is made for input and output in bases other than decimal. Numbers can be converted from decimal to octal by simply setting the output base to equal 8.

The actual limit on the number of digits that can be handled depends on the amount of storage available on the machine. Manipulation of numbers with many hundreds of digits is possible even on the smallest versions of UNIX.

The syntax of BC has been deliberately selected to agree substantially with the C language [2]. Those who are familiar with C will find few surprises in this language.

Simple Computations with Integers

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type in the line:

142857 + 285714

the program responds immediately with the line

428571

The operators –, *, /, %, and ^ can also be used; they indicate subtraction, multiplication, division, remaindering, and exponentiation, respectively. Division of integers produces an integer result truncated toward zero. Division by zero produces an error comment.

Any term in an expression may be prefixed by a minus sign to indicate that it is to be negated (the "unary" minus sign). The expression

7+–3

is interpreted to mean that –3 is to be added to 7.

More complex expressions with several operators and with parentheses are interpreted just as in Fortran, with ^ having the greatest binding power, then * and % and /, and finally + and –. Contents of parentheses are evaluated before material outside the parentheses. Exponentiations are performed from right to left and the other operators from left to right. The two expressions

a^b^c and a^(b^c)

are equivalent, as are the two expressions

a*b*c and (a*b)*c

BC shares with Fortran and C the undesirable convention that

a/b*c is equivalent to **(a/b)*c**

Internal storage registers to hold numbers have single lower-case letter names. The value of an expression can be assigned to a register in the usual way. The statement

```
x = x + 3
```

has the effect of increasing by three the value of the contents of the register named x. When, as in this case, the outermost operator is an =, the assignment is performed but the result is not printed. Only 26 of these named storage registers are available.

There is a built-in square root function whose result is truncated to an integer (but see scaling below). The lines

```
x = sqrt(191)
x
```

produce the printed result

```
13
```

Bases

There are special internal quantities, called "ibase" and "obase". The contents of "ibase", initially set to 10, determines the base used for interpreting numbers read in. For example, the lines

```
ibase = 8
11
```

will produce the output line

9

and you are all set up to do octal to decimal conversions. Beware, however of trying to change the input base back to decimal by typing

ibase = 10

Because the number 10 is interpreted as octal, this statement will have no effect. For those who deal in hexadecimal notation, the characters A–F are permitted in numbers (no matter what base is in effect) and are interpreted as digits having values 10–15 respectively. The statement

ibase = A

will change you back to decimal input base no matter what the current input base is. Negative and large positive input bases are permitted but useless. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

The contents of "obase", initially set to 10, are used as the base for output numbers. The lines

obase = 16
1000

will produce the output line

3E8

which is to be interpreted as a 3-digit hexadecimal number. Very large output bases are permitted, and they are sometimes useful. For example, large numbers can be output in groups of five digits by setting "obase" to 100000. Strange (i.e. 1, 0, or negative) output bases are handled appropriately.

Very large numbers are split across lines with 70 characters per line. Lines which are continued end with \. Decimal output conversion is practically instantaneous, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow. Non-decimal output conversion of a one hundred digit number takes about three seconds.

It is best to remember that "ibase" and "obase" have no effect whatever on the course of internal computation or on the evaluation of expressions, but only affect input and output conversion, respectively.

Scaling

A third special internal quantity called "scale" is used to determine the scale of calculated quantities. Numbers may have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations. We refer to the number of digits after the decimal point of a number as its scale.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules. For addition and subtraction, the scale of the result is the larger of the scales of the two operands. In this case, there is never any truncation of the result. For multiplications, the scale of the result is never less than the maximum of the two scales of the operands, never more than the sum of the scales of the operands and, subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity "scale". The scale of a quotient is the contents of the internal quantity "scale". The scale of a remainder is the sum of the scales of the quotient and the divisor. The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer. The scale of a square root is set to the maximum of the scale of the argument and the contents of "scale".

All of the internal operations are actually carried out in terms of integers, with digits being discarded when necessary. In every case where digits are discarded, truncation and not rounding is performed.

The contents of "scale" must be no greater than 99 and no less than 0. It is initially set to 0. In case you need more than 99 fraction digits, you may arrange your own scaling.

The internal quantities "scale", "ibase", and "obase" can be used in expressions just like other variables. The line

```
scale = scale + 1
```

increases the value of "scale" by one, and the line

```
scale
```

causes the current value of "scale" to be printed.

The value of "scale" retains its meaning as a number of decimal digits to be retained in internal computation even when "ibase" or "obase" are not equal to 10. The internal computations (which are

still conducted in decimal, regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal or octal or any other kind of digits.

Functions

The name of a function is a single lower-case letter. Function names are permitted to collide with simple variable names. Twenty-six different defined functions are permitted in addition to the twenty-six variable names. The line

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements, which make up the body of the function, ending with a right brace }. Return of control from a function occurs when a return statement is executed or when the end of the function is reached. The return statement can take either of the two forms

```
return
return(x)
```

In the first case, the value of the function is 0, and in the second, the value of the expression in parentheses.

Variables used in the function can be declared as automatic by a statement of the form

```
auto x,y,z
```

There can be only one "auto" statement in a function and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return. The values of any variables with the same names outside the function are not disturbed. Functions may be called recursively and the automatic variables at each level of call are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function with the single exception that they are given a value on entry to the function. An example of a function definition is

```
define a(x,y){
        auto z
        z = x*y
        return(z)
}
```

The value of this function, when called, will be the product of its two arguments.

A function is called by the appearance of its name followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions with no arguments are defined and called using parentheses with nothing between them: b().

If the function *a* above has been defined, then the line

```
a(7,3.14)
```

would cause the result 21.98 to be printed and the line

```
x = a(a(3,4),5)
```

would cause the value of x to become 60.

Subscripted Variables

A single lower-case letter variable name followed by an expression in brackets is called a subscripted variable (an array element). The variable name is called the array name and the expression in brackets is called the subscript. Only one-dimensional arrays are permitted. The names of arrays are permitted to collide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to zero and less than or equal to 2047.

Subscripted variables may be freely used in expressions, in function calls, and in return statements.

An array name may be used as an argument to a function, or may be declared as automatic in a function definition by the use of empty brackets:

```
f(a[ ])
define f(a[ ])
auto a[ ]
```

When an array name is so used, the whole contents of the array are copied for the use of the function, and thrown away on exit from the function. Array names which refer to whole arrays cannot be used in any other contexts.

Control Statements

The "if", the "while", and the "for" statements may be used to alter the flow within programs or to cause iteration. The range of each of them is a statement or a compound statement consisting of a collection of statements enclosed in braces. They are written in the following way

```
if(relation) statement
while(relation) statement
for(expression1; relation; expression2) statement
```

or

```
if(relation) {statements}
while(relation) {statements}
for(expression1; relation; expression2) {statements}
```

A relation in one of the control statements is an expression of the form

```
x>y
```

where two expressions are related by one of the six relational operators <, >, <=, >=, ==, or !=. The relation == stands for "equal to" and != stands for "not equal to". The meaning of the remaining relational operators is clear.

CAUTION

BEWARE of using = instead of == in a relational. Unfortunately, both of them are legal, so you will not get a diagnostic message, but = really will not do a comparison.

The "if" statement causes execution of its range if and only if the relation is true. Then control passes to the next statement in sequence.

The "while" statement causes execution of its range repeatedly as long as the relation is true. The relation is tested before each execution of its range and if the relation is false, control passes to the next statement beyond the range of the while.

The "for" statement begins by executing "expression1". Then the relation is tested and, if true, the statements in the range of the "for" are executed. Then "expression2" is executed. The relation is tested, and so on. The typical use of the "for" statement is for a controlled iteration, as in the statement

```
for(i=1; i<=10; i=i+1) i
```

which will print the integers from 1 to 10. Here are some examples of the use of the control statements.

```
define f(n){
auto i, x
x=1
for(i=1; i<=n; i=i+1) x=x*i
return(x)
}
```

The line

```
f(a)
```

will print *a* factorial if *a* is a positive integer. Here is the definition of a function which will compute values of the binomial coefficient (m and n are assumed to be positive integers).

```
define b(n,m){
auto x, j
x=1
for(j=1; j<=m; j=j+1) x=x*(n-j+1)/j
return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard for possible truncation errors:

```
scale = 20
define e(x){
        auto a, b, c, d, n
        a = 1
        b = 1
        c = 1
        d = 0
        n = 1
        while(1==1){
                a = a*x
                b = b*n
                c = c + a/b
                n = n + 1
                if(c==d) return(c)
                d = c
        }
}
```

Some Details

There are some language features that every user should know about even if he will not use them.

Normally statements are typed one to a line. It is also permissible to type several statements on a line separated by semicolons.

If an assignment statement is parenthesized, it then has a value and it can be used anywhere that an expression can. For example, the line

```
(x=y+17)
```

not only makes the indicated assignment, but also prints the resulting value.

Here is an example of a use of the value of an assignment statement even when it is not parenthesized.

```
x = a[i=i+1]
```

causes a value to be assigned to x and also increments i before it is used as a subscript.

The following constructs work in BC in exactly the same manner as they do in the C language. Consult the appendix or the C manuals [2] for their exact workings.

x=y=z is the same as	x=(y=z)
x =+ y	x = x+y
x =– y	x = x–y
x =* y	x = x*y
x =/ y	x = x/y
x =% y	x = x%y
x =^ y	x = x^y
x++	(x=x+1)–1
x—	(x=x–1)+1
++x	x = x+1
—x	x = x–1

Even if you don't intend to use the constructs, if you type one inadvertently, something correct but unexpected may happen.

CAUTION

WARNING! In some of these constructions, spaces are significant. There is a real difference between **x =– y** and **x= –y.** The first replaces x by x–y and the second by –y.

Three Important Things

1. To exit a BC program, type "quit".
2. There is a comment convention identical to that of C and of PL/I. Comments begin with '/*' and end with '*/'.
3. There is a library of math functions which may be obtained by typing at command level

   bc –l

   This command will load a set of library functions which, at the time of writing, consists of sine (named 's'), cosine ('c'), arctangent ('a'), natural logarithm ('l'), exponential ('e') and Bessel functions of integer order ('j(n,x)'). Doubtless more functions will be added in time. The library sets the scale to 20. You can reset it to something else if you like. The design of these mathematical library routines is discussed elsewhere [3].

If you type

bc file ...

BC will read and execute the named file or files before accepting commands from the keyboard. In this way, you may load your favorite programs and function definitions.

Acknowledgement

The compiler is written in YACC [4]; its original version was written by S. C. Johnson.

References

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

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[3] R. Morris, *A Library of Reference Standard Mathematical Subroutines*, Bell Laboratories internal memorandum, 1975.

[4] S. C. Johnson, *YACC — Yet Another Compiler-Compiler.* Bell Laboratories Computing Science Technical Report #32, 1978.

[5] R. Morris and L. L. Cherry, *DC — An Interactive Desk Calculator.*

Appendix

Notation

In the following pages syntactic categories are in *italics*; literals are in **bold**; material in brackets [ ] is optional.

Tokens

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs or comments. Newline characters or semicolons separate statements.

Comments

Comments are introduced by the characters /* and terminated by */.

Identifiers

There are three kinds of identifiers—ordinary identifiers, array identifiers and function identifiers. All three types consist of single lower-case letters. Array identifiers are followed by square brackets, possibly enclosing an expression describing a subscript. Arrays are singly dimensioned and may contain up to 2048 elements. Indexing begins at zero so an array may be indexed from 0 to 2047. Subscripts are truncated to integers. Function identifiers are followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict; a program can have a variable named **x**, an array named **x** and a function named **x**, all of which are separate and distinct.

Keywords

The following are reserved keywords:

ibase	**if**
obase	**break**
scale	**define**
sqrt	**auto**
length	**return**
while	**quit**
for	

Constants

Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits **A–F** are also recognized as digits with values 10–15, respectively.

Expressions

The value of an expression is printed unless the main operator is an assignment. Precedence is the same as the order of presentation here, with highest appearing first. Left or right associativity, where applicable, is discussed with each operator.

Primitive Expressions

Named Expressions

Named expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.

identifiers

Simple identifiers are named expressions. They have an initial value of zero.

array-name [*expression*]

Array elements are named expressions. They have an initial value of zero.

scale, **ibase** and **obase**

The internal registers **scale**, **ibase** and **obase** are all named expressions. **scale** is the number of digits after the decimal point to be retained in arithmetic operations. **scale** has an initial value of zero. **ibase** and **obase** are the input and output number radix respectively. Both **ibase** and **obase** have initial values of 10.

Function Calls

function-name ([*expression* [, *expression* . . .]])

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a return statement, the value of the function is the value of the expression in the parentheses of the return statement or is zero if no expression is provided or if there is no return statement.

sqrt (*expression*)

The result is the square root of the expression. The result is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of **scale,** whichever is larger.

length (*expression*)

The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

scale (*expression*)

The result is the scale of the expression. The scale of the result is zero.

Constants

Constants are primitive expressions.

Parentheses

An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

Unary Operators

The unary operators bind right to left.

– expression

The result is the negative of the expression.

++ named-expression

The named expression is incremented by one. The result is the value of the named expression after incrementing.

— named-expression

The named expression is decremented by one. The result is the value of the named expression after decrementing.

named-expression ++

The named expression is incremented by one. The result is the value of the named expression before incrementing.

named-expression —

The named expression is decremented by one. The result is the value of the named expression before decrementing.

Exponentiation Operator

The exponentiation operator binds right to left.

expression ^ *expression*

The result is the first expression raised to the power of the second expression. The second expression must be an integer. If a is the scale of the left expression and b is the absolute value of the right expression, then the scale of the result is:

$$\min(a \times b, \max(\textbf{scale}, a))$$

Multiplicative Operators

The operators *, /, % bind left to right.

expression * *expression*

The result is the product of the two expressions. If a and b are the scales of the two expressions, then the scale of the result is:

$$\min(a+b, \max(\textbf{scale}, a, b))$$

expression / *expression*

The result is the quotient of the two expressions. The scale of the result is the value of **scale**.

expression % *expression*

The % operator produces the remainder of the division of the two expressions. More precisely, *a%b* is *a–a/b*b*. The scale of the result is the sum of the scale of the divisor and the value of **scale**.

Additive Operators

The additive operators bind left to right.

expression + *expression*

The result is the sum of the two expressions. The scale of the result is the maximun of the scales of the expressions.

expression – expression

The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

Assignment Operators

The assignment operators bind right to left.

named-expression = expression

This expression results in assigning the value of the expression on the right to the named expression on the left.

named-expression =+ expression

named-expression =– expression

named-expression = expression*

named-expression =/ expression

named-expression =% expression

named-expression =^ expression

The result of the above expressions is equivalent to "named expression = named expression OP expression", where OP is the operator after the = sign.

Relations

Unlike all other operators, the relational operators are only valid as the object of an **if**, **while**, or inside a **for** statement.

expression < expression

expression > expression

expression <= expression

expression >= expression

expression == expression

expression != expression

Storage Classes

There are only two storage classes in BC, global and automatic (local). Only identifiers that are to be local to a function need be declared with the **auto** command. The arguments to a function are local to the function. All other identifiers are assumed to be global and available to all functions. All identifiers, global and local, have initial values of zero. Identifiers declared as **auto** are allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls. **auto** arrays are specified by the array name followed by empty square brackets.

Automatic variables in BC do not work in exactly the same way as in either C or PL/I. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

Statements

Statements must be separated by semicolon or newline. Except where altered by control statements, execution is sequential.

Expression Statements

When a statement is an expression, unless the main operator is an assignment, the value of the expression is printed, followed by a newline character.

Compound Statements

Statements may be grouped together and used when one statement is expected by surrounding them with { }.

Quoted String Statements

"any string"

This statement prints the string inside the quotes.

If Statements

if (*relation*) *statement*

The substatement is executed if the relation is true.

While Statements

while (*relation*) *statement*

The statement is executed while the relation is true. The test occurs before each execution of the statement.

For Statements

for (*expression*; *relation*; *expression*) *statement*

The for statement is the same as

```
first-expression
while (relation) {
        statement
        last-expression
}
```

All three expressions must be present.

Break Statements

break

break causes termination of a **for** or **while** statement.

Auto Statements

auto *identifier* [,*identifier*]

The auto statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The auto statement must be the first statement in a function definition.

Define Statements

define([*parameter* [,*parameter* . . .]]) {
statements }

The define statement defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

Return Statements

return

return(*expression*)

The return statement causes termination of a function, popping of its auto variables, and specifies the result of the function. The first form is equivalent to **return(0)**. The result of the function is the result of the expression in parentheses.

Quit

The quit statement stops execution of a BC program and returns control to UNIX when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an **if**, **for**, or **while** statement.

3

DC — An Interactive Desk Calculator

Abstract

DC is an interactive desk calculator program implemented on the UNIX time-sharing system to do arbitrary-precision integer arithmetic. It has provision for manipulating scaled fixed-point numbers and for input and output in bases other than decimal.

The size of numbers that can be manipulated is limited only by available core storage. On typical implementations of UNIX , the size of numbers that can be handled varies from several hundred digits on the smallest systems to several thousand on the largest.

Introduction

DC is an arbitrary precision arithmetic package implemented on the UNIX time-sharing system in the form of an interactive desk calculator. It works like a stacking calculator using reverse Polish notation. Ordinarily DC operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained.

A language called BC [1] has been developed which accepts programs written in the familiar style of higher-level programming languages and compiles output which is interpreted by DC. Some of the commands described below were designed for the compiler interface and are not easy for a human user to manipulate.

Source: Robert Morris and Lorinda Cherry, *DC — An Interactive Desk Calculator* (Murray Hill, N.J.: Bell Laboratories, 1978).

Numbers that are typed into DC are put on a push-down stack. DC commands work by taking the top number or two off the stack, performing the desired operation, and pushing the result on the stack. If an argument is given, input is taken from that file until its end, then from the standard input.

Synoptic Description

Here we describe the DC commands that are intended for use by people. The additional commands that are intended to be invoked by compiled output are described in the detailed description.

Any number of commands are permitted on a line. Blanks and new-line characters are ignored except within numbers and in places where a register name is expected.

The following constructions are recognized:

number The value of the number is pushed onto the main stack. A number is an unbroken string of the digits 0–9 and the capital letters A–F which are treated as digits with values 10–15 respectively. The number may be preceded by an underscore to input a negative number. Numbers may contain decimal points.

+ – * / % ^ The top two values on the stack are added (+), subtracted (–), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. The result of a division is an integer truncated toward zero. See the detailed description below for the treatment of numbers with decimal points. An exponent must not have any digits after the decimal point.

s*x* The top of the main stack is popped and stored into a register named *x*, where *x* may be any character. If the s is capitalized, *x* is treated as a stack and the value is pushed onto it. Any character, even blank or new-line, is a valid register name.

l*x* The value in register *x* is pushed onto the stack. The register *x* is not altered. If the l is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

All registers start with empty value which is treated as a zero by the command **l** and is treated as an error by the command **L**.

d The top value on the stack is duplicated.

p The top value on the stack is printed. The top value remains unchanged.

f All values on the stack and in registers are printed.

x treats the top element of the stack as a character string, removes it from the stack, and executes it as a string of DC commands.

[...] puts the bracketed character string onto the top of the stack.

q exits the program. If executing a string, the recursion level is popped by two. If **q** is capitalized, the top value on the stack is popped and the string execution level is popped by that value.

<*x* >*x* =*x* !<*x* !>*x* !=*x*

The top two elements of the stack are popped and compared. Register *x* is executed if they obey the stated relation. Exclamation point is negation.

v replaces the top element on the stack by its square root. The square root of an integer is truncated to an integer. For the treatment of numbers with decimal points, see the detailed description below.

! interprets the rest of the line as a UNIX command. Control returns to DC when the UNIX command terminates.

c All values on the stack are popped; the stack becomes empty.

i The top value on the stack is popped and used as the number radix for further input. If **i** is capitalized, the value of the input base is pushed onto the stack. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 or greater than 16.

o The top value on the stack is popped and used as the number radix for further output. If **o** is capitalized, the value of the output base is pushed onto the stack.

k The top of the stack is popped, and that value is used as a scale factor that influences the number of decimal places that are maintained during multiplication, division, and exponentiation. The scale factor must be greater than or equal to zero and less than 100. If **k** is capitalized, the value of the scale factor is pushed onto the stack.

z The value of the stack level is pushed onto the stack.

? A line of input is taken from the input source (usually the console) and executed.

Detailed Description

Internal Representation of Numbers

Numbers are stored internally using a dynamic storage allocator. Numbers are kept in the form of a string of digits to the base 100 stored one digit per byte (centennial digits). The string is stored with the low-order digit at the beginning of the string. For example, the representation of 157 is 57,1. After any arithmetic operation on a number, care is taken that all digits are in the range 0–99 and that the number has no leading zeros. The number zero is represented by the empty string.

Negative numbers are represented in the 100's complement notation, which is analogous to two's complement notation for binary numbers. The high order digit of a negative number is always –1 and all other digits are in the range 0–99. The digit preceding the high order –1 digit is never a 99. The representation of –157 is 43,98,–1. We shall call this the canonical form of a number. The advantage of this kind of representation of negative numbers is ease of addition. When addition is performed digit by digit, the result is formally correct. The result need only be modified, if necessary, to put it into canonical form.

Because the largest valid digit is 99 and the byte can hold numbers twice that large, addition can be carried out and the handling of carries done later when that is convenient, as it sometimes is.

An additional byte is stored with each number beyond the high order digit to indicate the number of assumed decimal digits after the decimal point. The representation of .001 is 1,*3* where the scale has

been italicized to emphasize the fact that it is not the high order digit. The value of this extra byte is called the **scale factor** of the number.

The Allocator

DC uses a dynamic string storage allocator for all of its internal storage. All reading and writing of numbers internally is done through the allocator. Associated with each string in the allocator is a four-word header containing pointers to the beginning of the string, the end of the string, the next place to write, and the next place to read. Communication between the allocator and DC is done via pointers to these headers.

The allocator initially has one large string on a list of free strings. All headers except the one pointing to this string are on a list of free headers. Requests for strings are made by size. The size of the string actually supplied is the next higher power of 2. When a request for a string is made, the allocator first checks the free list to see if there is a string of the desired size. If none is found, the allocator finds the next larger free string and splits it repeatedly until it has a string of the right size. Left-over strings are put on the free list. If there are no larger strings, the allocator tries to coalesce smaller free strings into larger ones. Since all strings are the result of splitting large strings, each string has a neighbor that is next to it in core and, if free, can be combined with it to make a string twice as long. This is an implementation of the "buddy system" of allocation described in [2].

Failing to find a string of the proper length after coalescing, the allocator asks the system for more space. The amount of space on the system is the only limitation on the size and number of strings in DC. If at any time in the process of trying to allocate a string, the allocator runs out of headers, it also asks the system for more space.

There are routines in the allocator for reading, writing, copying, rewinding, forward-spacing, and backspacing strings. All string manipulation is done using these routines.

The reading and writing routines increment the read pointer or write pointer so that the characters of a string are read or written in succession by a series of read or write calls. The write pointer is interpreted as the end of the information-containing portion of a string and a call to read beyond that point returns an end-of-string indication. An attempt to write beyond the end of a string causes the allocator to

allocate a larger space and then copy the old string into the larger block.

Internal Arithmetic

All arithmetic operations are done on integers. The operands (or operand) needed for the operation are popped from the main stack and their scale factors stripped off. Zeros are added or digits removed as necessary to get a properly scaled result from the internal arithmetic routine. For example, if the scale of the operands is different and decimal alignment is required, as it is for addition, zeros are appended to the operand with the smaller scale. After performing the required arithmetic operation, the proper scale factor is appended to the end of the number before it is pushed on the stack.

A register called **scale** plays a part in the results of most arithmetic operations. **scale** is the bound on the number of decimal places retained in arithmetic computations. **scale** may be set to the number on the top of the stack truncated to an integer with the **k** command. **K** may be used to push the value of **scale** on the stack. **scale** must be greater than or equal to 0 and less than 100. The descriptions of the individual arithmetic operations will include the exact effect of **scale** on the computations.

Addition and Subtraction

The scales of the two numbers are compared and trailing zeros are supplied to the number with the lower scale to give both numbers the same scale. The number with the smaller scale is multiplied by 10 if the difference of the scales is odd. The scale of the result is then set to the larger of the scales of the two operands.

Subtraction is performed by negating the number to be subtracted and proceeding as in addition.

Finally, the addition is performed digit by digit from the low order end of the number. The carries are propagated in the usual way. The resulting number is brought into canonical form, which may require stripping of leading zeros, or for negative numbers replacing the high-order configuration 99,–1 by the digit –1. In any case, digits which are not in the range 0–99 must be brought into that range, propagating any carries or borrows that result.

Multiplication

The scales are removed from the two operands and saved. The operands are both made positive. Then multiplication is performed in a digit by digit manner that exactly mimics the hand method of multiplying. The first number is multiplied by each digit of the second number, beginning with its low order digit. The intermediate products are accumulated into a partial sum which becomes the final product. The product is put into the canonical form and its sign is computed from the signs of the original operands.

The scale of the result is set equal to the sum of the scales of the two operands. If that scale is larger than the internal register **scale** and also larger than both of the scales of the two operands, then the scale of the result is set equal to the largest of these three last quantities.

Division

The scales are removed from the two operands. Zeros are appended or digits removed from the dividend to make the scale of the result of the integer division equal to the internal quantity **scale**. The signs are removed and saved.

Division is performed much as it would be done by hand. The difference of the lengths of the two numbers is computed. If the divisor is longer than the dividend, zero is returned. Otherwise the top digit of the divisor is divided into the top two digits of the dividend. The result is used as the first (high-order) digit of the quotient. It may turn out be one unit too low, but if it is, the next trial quotient will be larger than 99 and this will be adjusted at the end of the process. The trial digit is multiplied by the divisor and the result subtracted from the dividend and the process is repeated to get additional quotient digits until the remaining dividend is smaller than the divisor. At the end, the digits of the quotient are put into the canonical form, with propagation of carry as needed. The sign is set from the sign of the operands.

Remainder

The division routine is called and division is performed exactly as described. The quantity returned is the remains of the dividend at the end of the divide process. Since division truncates toward zero, remainders have the same sign as the dividend. The scale of the remainder is set to the maximum of the scale of the dividend and the scale of the quotient plus the scale of the divisor.

Square Root

The scale is stripped from the operand. Zeros are added if necessary to make the integer result have a scale that is the larger of the internal quantity **scale** and the scale of the operand.

The method used to compute sqrt(y) is Newton's method with successive approximations by the rule

$$x_{n+1} = \tfrac{1}{2}(x_n + \frac{y}{x_n})$$

The initial guess is found by taking the integer square root of the top two digits.

Exponentiation

Only exponents with zero scale factor are handled. If the exponent is zero, then the result is 1. If the exponent is negative, then it is made positive and the base is divided into one. The scale of the base is removed.

The integer exponent is viewed as a binary number. The base is repeatedly squared and the result is obtained as a product of those powers of the base that correspond to the positions of the one-bits in the binary representation of the exponent. Enough digits of the result are removed to make the scale of the result the same as if the indicated multiplication had been performed.

Input Conversion and Base

Numbers are converted to the internal representation as they are read in. The scale stored with a number is simply the number of fractional digits input. Negative numbers are indicated by preceding the number with a _ (an underscore). The hexadecimal digits A–F correspond to the numbers 10–15 regardless of input base. The **i** command can be used to change the base of the input numbers. This command pops the stack, truncates the resulting number to an integer, and uses it as the input base for all further input. The input base is initialized to 10 but may, for example be changed to 8 or 16 to do octal or hexadecimal to decimal conversions. The command **I** will push the value of the input base on the stack.

Output Commands

The command **p** causes the top of the stack to be printed. It does not remove the top of the stack. All of the stack and internal registers can be output by typing the command **f**. The **o** command can be used to change the output base. This command uses the top of the stack, truncated to an integer as the base for all further output. The output base in initialized to 10. It will work correctly for any base. The command **O** pushes the value of the output base on the stack.

Output Format and Base

The input and output bases only affect the interpretation of numbers on input and output; they have no effect on arithmetic computations. Large numbers are output with 70 characters per line; a \ indicates a continued line. All choices of input and output bases work correctly, although not all are useful. A particularly useful output base is 100000, which has the effect of grouping digits in fives. Bases of 8 and 16 can be used for decimal-octal or decimal-hexadecimal conversions.

Internal Registers

Numbers or strings may be stored in internal registers or loaded on the stack from registers with the commands **s** and **l**. The command **s***x* pops the top of the stack and stores the result in register **x**. *x* can be any character. **l***x* puts the contents of register **x** on the top of the stack. The **l** command has no effect on the contents of register *x*. The **s** command, however, is destructive.

Stack Commands

The command **c** clears the stack. The command **d** pushes a duplicate of the number on the top of the stack on the stack. The command **z** pushes the stack size on the stack. The command **X** replaces the number on the top of the stack with its scale factor. The command **Z** replaces the top of the stack with its length.

Subroutine Definitions and Calls

Enclosing a string in **[ ]** pushes the ascii string on the stack. The **q** command quits or in executing a string, pops the recursion levels by two.

Internal Registers – Programming DC

The load and store commands together with **[ ]** to store strings, **x** to execute and the testing commands '<', '>', '=', '!<', '!>', '!=' can be used to program DC. The **x** command assumes the top of the stack is an string of DC commands and executes it. The testing commands compare the top two elements on the stack and if the relation holds, execute the register that follows the relation. For example, to print the numbers 0-9,

```
[lip1+ si li10>a]sa
0si lax
```

Push-down Registers and Arrays

These commands were designed for used by a compiler, not by people. They involve push-down registers and arrays. In addition to the stack that commands work on, DC can be thought of as having individual stacks for each register. These registers are operated on by the commands **S** and **L**. **S***x* pushes the top value of the main stack onto the stack for the register *x*. **L***x* pops the stack for register *x* and puts the result on the main stack. The commands **s** and **l** also work on registers but not as push-down stacks. **l** doesn't effect the top of the register stack, and **s** destroys what was there before.

The commands to work on arrays are **:** and **;**. **:***x* pops the stack and uses this value as an index into the array *x*. The next element on the stack is stored at this index in *x*. An index must be greater than or equal to 0 and less than 2048. **;***x* is the command to load the main stack from the array *x*. The value on the top of the stack is the index into the array *x* of the value to be loaded.

Miscellaneous Commands

The command **!** interprets the rest of the line as a UNIX command and passes it to UNIX to execute. One other compiler command is **Q**. This command uses the top of the stack as the number of levels of recursion to skip.

Design Choices

The real reason for the use of a dynamic storage allocator was that a general purpose program could be (and in fact has been) used for a variety of other tasks. The allocator has some value for input and for compiling (i.e. the bracket [...] commands) where it cannot be known in advance how long a string will be. The result was that at a modest cost in execution time, all considerations of string allocation and sizes of strings were removed from the remainder of the program and debugging was made easier. The allocation method used wastes approximately 25% of available space.

The choice of 100 as a base for internal arithmetic seemingly has no compelling advantage. Yet the base cannot exceed 127 because of

hardware limitations and at the cost of 5% in space, debugging was made a great deal easier and decimal output was made much faster.

The reason for a stack-type arithmetic design was to permit all DC commands from addition to subroutine execution to be implemented in essentially the same way. The result was a considerable degree of logical separation of the final program into modules with very little communication between modules.

The rationale for the lack of interaction between the scale and the bases was to provide an understandable means of proceeding after a change of base or scale when numbers had already been entered. An earlier implementation which had global notions of scale and base did not work out well. If the value of **scale** were to be interpreted in the current input or output base, then a change of base or scale in the midst of a computation would cause great confusion in the interpretation of the results. The current scheme has the advantage that the value of the input and output bases are only used for input and output, respectively, and they are ignored in all other operations. The value of scale is not used for any essential purpose by any part of the program and it is used only to prevent the number of decimal places resulting from the arithmetic operations from growing beyond all bounds.

The design rationale for the choices for the scales of the results of arithmetic were that in no case should any significant digits be thrown away if, on appearances, the user actually wanted them. Thus, if the user wants to add the numbers 1.5 and 3.517, it seemed reasonable to give him the result 5.017 without requiring him to unnecessarily specify his rather obvious requirements for precision.

On the other hand, multiplication and exponentiation produce results with many more digits than their operands and it seemed reasonable to give as a minimum the number of decimal places in the operands but not to give more than that number of digits unless the user asked for them by specifying a value for **scale**. Square root can be handled in just the same way as multiplication. The operation of division gives arbitrarily many decimal places and there is simply no way to guess how many places the user wants. In this case only, the user must specify a **scale** to get any decimal places at all.

The scale of remainder was chosen to make it possible to recreate the dividend from the quotient and remainder. This is easy to implement; no digits are thrown away.

References

[1] L. L. Cherry, R. Morris, *BC – An Arbitrary Precision Desk-Calculator Language*.

[2] K. C. Knowlton, *A Fast Storage Allocator,* Comm. ACM **8**, pp. 623-625 (Oct. 1965).

4

An Introduction to Display Editing with VI

Introduction

Vi (visual) is a display oriented interactive text editor. When using *vi* the screen of your terminal acts as a window into the file which you are editing. Changes which you make to the file are reflected in what you see.

Using *vi* you can insert new text any place in the file quite easily. Most of the commands to *vi* move the cursor around in the file. There are commands to move the cursor forward and backward in units of characters, words, sentences and paragraphs. A small set of operators, like **d** for delete and **c** for change, are combined with the motion commands to form operations such as delete word or change paragraph, in a simple and natural way. This regularity and the mnemonic assignment of commands to keys makes the editor command set easy to remember and to use.

Vi will work on a large number of display terminals, and new terminals are easily driven after editing a terminal description file. While it is advantageous to have an intelligent terminal which can locally insert and delete lines and characters from the display, the editor will function quite well on dumb terminals over slow phone lines. The editor makes allowance for the low bandwidth in these situations and uses smaller window sizes and different display updating algorithms to make best use of the limited speed available.

It is also possible to use the command set of *vi* on hardcopy terminals, storage tubes and "glass tty's" using a one line editing window; thus *vi's* command set is available on all terminals. The full command set

---

Source: William Joy and Mark Horton, *An Introduction to Display Editing with VI* (Berkeley, CA: University of California).

of the more traditional, line oriented editor *ex* is available within *vi;* it is quite simple to switch between the two modes of editing.

1. Getting Started

This document provides a quick introduction to *vi*. (Pronounced *vee-eye*.) You should be running *vi* on a file you are familiar with while you are reading this. The first part of this document (sections 1 through 5) describes the basics of using *vi*. Some topics of special interest are presented in section 6, and some nitty-gritty details of how the editor functions are saved for section 7 to avoid cluttering the presentation here.

There is also a short appendix here, which gives for each character the special meanings which this character has in *vi*. Attached to this document should be a quick reference card. This card summarizes the commands of *vi* in a very compact format. You should have the card handy while you are learning *vi*.

1.1 Specifying Terminal Type

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

Code	Full Name	Type
2621	Hewlett-Packard 2621A/P	Intelligent
2645	Hewlett-Packard 264x	Intelligent
act4	Microterm ACT-IV	Dumb
act5	Microterm ACT-V	Dumb
adm3a	Lear Siegler ADM-3a	Dumb
adm31	Lear Siegler ADM-31	Intelligent
c100	Human Design Concept 100	Intelligent
dm1520	Datamedia 1520	Dumb
dm2500	Datamedia 2500	Intelligent
dm3025	Datamedia 3025	Intelligent
fox	Perkin-Elmer Fox	Dumb

h1500	Hazeltine 1500	Intelligent
h19	Heathkit h19	Intelligent
i100	Infoton 100	Intelligent
mime	Imitating a smart act4	Intelligent
t1061	Teleray 1061	Intelligent
vt52	Dec VT-52	Dumb

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

% **setenv TERM** 2621

This command works with the *csh* shell. If you are using the standard Bourne shell *sh* then you should give the commands

$ **TERM**=2621
$ **export TERM**

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

setenv TERM `**tset** – –d mime`

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

TERM=`**tset** – –d mime`

Tset knows which terminals are hardwired to each port and needs only to be told that when you dial in you are probably on a *mime*. *Tset* is usually used to change the erase and kill characters, too.

1.2 Editing a File

After telling the system which kind of terminal you have, you should make a copy of a file you are familiar with, and run *vi* on this file, giving the command

% **vi** *name*

replacing *name* with the name of the copy file you just created. The screen should clear and the text of your file should appear on the screen. If something else happens refer to the footnote.[1]

1.3 The Editor's Copy: The Buffer

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

1.4 Notational Conventions

In our examples, input which must be typed as is will be presented in **bold face**. Text which should be replaced with appropriate input will be given in *italics*. We will represent special characters in SMALL CAPITALS.

1.5 Arrow Keys

The editor command set is independent of the terminal you are using. On most terminals with cursor positioning keys, these keys will also work within the editor. If you don't have cursor positioning keys, or even if you do, you can use the **h j k** and **l** keys as cursor positioning keys (these are labelled with arrows on an *adm3a*).[2]

---

1. If you gave the system an incorrect terminal type code then the editor may have just made a mess out of your screen. This happens when it sends control codes for one kind of terminal to some other kind of terminal. In this case hit the keys **:q** (colon and the q key) and then hit the RETURN key. This should get you back to the command level interpreter. Figure out what you did wrong (ask someone else if necessary) and try again.

   Another thing which can go wrong is that you typed the wrong file name and the editor just printed an error diagnostic. In this case you should follow the above procedure for getting out of the editor, and try again this time spelling the file name correctly.

   If the editor doesn't seem to respond to the commands which you type here, try sending an interrupt to it by hitting the DEL or RUB key on your terminal, and then hitting the **:q** command again followed by a carriage return.
2. As we will see later, *h* moves back to the left (like control-h which is a backspace), *j* moves down (in the same column), *k* moves up (in the same column), and *l* moves to the right.

(Particular note for the HP2621: on this terminal the function keys must be *shifted* (ick) to send to the machine, otherwise they only act locally. Unshifted use will leave the cursor positioned incorrectly.)

1.6 Special Characters: ESC, CR and DEL

Several of these special characters are very important, so be sure to find them right now. Look on your keyboard for a key labelled ESC or ALT. It should be near the upper left corner of your terminal. Try hitting this key a few times. The editor will ring the bell to indicate that it is in a quiescent state.[3] Partially formed commands are cancelled by ESC, and when you insert text in the file you end the text insertion with ESC. This key is a fairly harmless one to hit, so you can just hit it if you don't know what is going on until the editor rings the bell.

The CR or RETURN key is important because it is used to terminate certain commands. It is usually at the right side of the keyboard, and is the same command used at the end of each shell command.

Another very useful key is the DEL or RUB key, which generates an interrupt, telling the editor to stop what it is doing. It is a forceful way of making the editor listen to you, or to return it to the quiescent state if you don't know or don't like what is going on. Try hitting the '/' key on your terminal. This key is used when you want to specify a string to be searched for. The cursor should now be positioned at the bottom line of the terminal after a '/' printed as a prompt. You can get the cursor back to the current position by hitting the DEL or RUB key; try this now.[4] From now on we will simply refer to hitting the DEL or RUB key as "sending an interrupt."[5]

The editor often echoes your commands on the last line of the terminal. If the cursor is on the first position of this last line, then the editor is performing a computation, such as computing a new position in the file after a search or running a command to reformat part of the

3. On smart terminals where it is possible, the editor will quietly flash the screen rather than ringing the bell.
4. Backspacing over the '/' will also cancel the search.
5. On some systems, this interruptibility comes at a price: you cannot type ahead when the editor is computing with the cursor on the bottom line.

buffer. When this is happening you can stop the editor by sending an interrupt.

1.7 Getting Out of the Editor

After you have worked with this introduction for a while, and you wish to do something else, you can give the command **ZZ** to the editor. This will write the contents of the editor's buffer back into the file you are editing, if you made any changes, and then quit from the editor. You can also end an editor session by giving the command **:q!**CR;[6] this is a dangerous but occasionally essential command which ends the editor session and discards all your changes. You need to know about this command in case you change the editor's copy of a file you wish only to look at. Be very careful not to give this command when you really want to save the changes you have made.

2. Moving Around in the File

2.1 Scrolling and Paging

The editor has a number of commands for moving around in the file. The most useful of these is generated by hitting the control and D keys at the same time, a control-D or '^D'. We will use this two character notation for referring to these control keys from now on. You may have a key labelled '^' on your terminal. This key will be represented as '↑' in this document; '^' is exclusively used as part of the '^x' notation for control characters.[7]

As you know now if you tried hitting **^D**, this command scrolls down in the file. The **D** thus stands for down. Many editor commands are mnemonic and this makes them much easier to remember. For instance the command to scroll up is **^U**. Many dumb terminals can't

6. All commands which read from the last display line can also be terminated with a ESC as well as an CR.
7. If you don't have a '^' key on your terminal then there is probably a key labelled '↑'; in any case these characters are one and the same.

scroll up at all, in which case hitting ^U clears the screen and refreshes it with a line which is farther back in the file at the top.

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

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

Notice the difference between scrolling and paging. If you are trying to read the text in a file, hitting ^F to move forward a page will leave you only a little context to look back at. Scrolling on the other hand leaves more context, and happens more smoothly. You can continue to read the text as scrolling is taking place.

2.2 Searching, Goto, and Previous Context

Another way to position yourself in the file is by giving the editor a string to search for. Type the character / followed by a string of characters terminated by CR. The editor will position the cursor at the next occurrence of this string. Try hitting **n** to then go to the next occurrence of this string. The character **?** will search backwards from where you are, and is otherwise like /.[8]

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

If you wish the search to match only at the beginning of a line, begin the search string with an ↑. To match only at the end of a line, end the search string with a **$**. Thus /↑**search**CR will search for the word

8. These searches will normally wrap around the end of the file, and thus find the string even if it is not on a line in the direction you search provided it is anywhere else in the file. You can disable this wraparound in scans by giving the command **:se nowrapscan**CR, or more briefly **:se nows**CR.

"search" at the beginning of a line, and /last$CR searches for the word "last" at the end of a line.[9]

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

If you are near the end of the file, and the last line is not at the bottom of the screen, the editor will place only the character '~' on each remaining line. This indicates that the last line in the file is on the screen; that is, the '~' lines are past the end of the file.

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

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

2.3 Moving Around on the Screen

Now try just moving the cursor around on the screen. If your terminal has arrow keys (4 or 5 keys with arrows going in each direction) try them and convince yourself that they work. If you don't have working arrow keys, you can always use **h, j, k,** and **l** . Experienced users of *vi* prefer these keys to arrow keys, because they are usually right underneath their fingers.

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

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

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

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

2.4 Moving within a Line

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

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

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

2.5 Summary

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

2.6 View

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

3. Making Simple Changes

3.1 Inserting

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

Now try finding a word which can, but does not, end in an 's'. Position yourself at this word and type **e** (move to end of word), then **a** for append and then 'sESC' to terminate the textual insert. This sequence of commands can be used to easily pluralize a word.

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

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

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

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

While you are inserting new text, you can use the characters you normally use at the system command level (usually **^H** or **#**) to

backspace over the last character which you typed, and the character which you use to kill input lines (usually @, ^X, or ^U) to erase the input you have typed on the current line.[10] The character ^W will erase a whole word and leave you after the space after the previous word; it is useful for quickly backing up in an insert.

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

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

3.2 Making Small Corrections

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

If the character is incorrect, you can replace it with the correct character by giving the command **r***c*, where *c* is replaced by the correct character. Finally if the character which is incorrect should be replaced by more than one character, give the command **s** which substitutes a string of characters, ending with ESC, for it. If there are a small number of characters which are wrong you can precede **s** with a

10. In fact, the character ^H (backspace) always works to erase the last input character here, regardless of what your erase character is.

count of the number of characters to be replaced. Counts are also useful with **x** to specify the number of characters to be deleted.

3.3 More Corrections: Operators

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

Now try **db**. This deletes a word backwards, namely the preceding word. Try **d**SPACE. This deletes a single character, and is equivalent to the **x** command.

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

3.4 Operating on Lines

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

Try repeating the **c** operator twice; this will change a whole line, erasing its previous contents and replacing them with text you type up to an ESC.[11]

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

You can delete or change more than one line by preceding the **dd** or **cc** with a count, i.e. **5dd** deletes 5 lines. You can also give a command like **dL** to delete all the lines up to and including the last line on the screen, or **d3L** to delete through the third from the bottom line. Try some commands like this now.[12] Notice that the editor lets you know when you change a large number of lines so that you can see the extent of the change. The editor will also always tell you when a change you make affects text which you cannot see.

3.5 Undoing

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

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

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

3.6 Summary

SPACE	advance the cursor one position
^H	backspace the cursor
^W	erase a word during an insert
erase	your erase (usually ^H or #), erases a character during an insert
kill	your kill (usually @, ^X, or ^U), kills the insert on this

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

	line
.	repeats the changing command
O	opens and inputs new lines, above the current
U	undoes the changes you made to the current line
a	appends text after the cursor
c	changes the object you specify to the following text
d	deletes the object you specify
i	inserts text before the cursor
o	opens and inputs new lines, below the current
u	undoes the last change

4. Moving About, Rearranging and Duplicating Text

4.1 Low Level Character Motions

Now move the cursor to a line where there is a punctuation or a bracketing character such as a parenthesis or a comma or period. Try the command **f***x* where *x* is this character. This command finds the next *x* character to the right of the cursor in the current line. Try then hitting a **;**, which finds the next instance of the same character. By using the **f** command and then a sequence of **;**'s you can often get to a particular place in a line much faster than with a sequence of word motions or SPACEs. There is also a **F** command, which is like **f**, but searches backward. The **;** command repeats **F** also.

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

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

Your file may have tab (**^I**) characters in it. These characters are represented as a number of spaces expanding to a tab stop, where tab stops are every 8 positions.[13] When the cursor is at a tab, it sits on

the last of the several spaces which represent that tab. Try moving the cursor back and forth over tabs so you understand how this works.

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

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

4.2 Higher Level Text Objects

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

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

The operations { and } move over paragraphs and the operations [[and]] move over sections.[14]

13. This is settable by a command of the form **:se ts=***x*CR, where *x* is 4 to set tabstops every four columns. This has effect on the screen representation within the editor.
14. The [[and]] operations require the operation character to be doubled because they can move the cursor far from where it currently is. While it is easy to get back with the command ``, these commands would still be frustrating if they were easy to hit accidentally.

A paragraph begins after each empty line, and also at each of a set of paragraph macros, specified by the pairs of characters in the definition of the string valued option *paragraphs*. The default setting for this option defines the paragraph macros of the *–ms* and *–mm* macro packages, i.e. the '.IP', '.LP', '.PP' and '.QP', '.P' and '.LI' macros.[15] Each paragraph boundary is also a sentence boundary. The sentence and paragraph commands can be given counts to operate over groups of sentences and paragraphs.

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

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

4.3 Rearranging and Duplicating Text

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

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

15. You can easily change or extend this set of macros by assigning a different string to the *paragraphs* option in your EXINIT. See section 6.2 for details. The '.bp' directive is also considered to start a paragraph.

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

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

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

4.4 Summary

↑	first non-white on line
$	end of line
)	forward sentence
}	forward paragraph
]]	forward section
(	backward sentence
{	backward paragraph
[[	backward section
f*x*	find *x* forward in line
p	put text back, after cursor or below current line
y	yank operator, for copies and moves

t*x*	up to *x* forward, for operators
F*x*	f backward in line
P	put text back, before cursor or above current line
T*x*	t backward in line

5. High Level Commands

5.1 Writing, Quitting, Editing New Files

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

If you have changed the editor's copy of the file but do not wish to save your changes, either because you messed up the file or decided that the changes are not an improvement to the file, then you can give the command **:q!**CR to quit from the editor without writing the changes. You can also reedit the same file (starting over) by giving the command **:e!**CR. These commands should be used only rarely, and with caution, as it is not possible to recover the changes you have made after you discard them in this manner.

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

5.2 Escaping to a Shell

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

can also give another : command when it asks you for a RETURN; in this case the screen will not be redrawn.

If you wish to execute more than one command in the shell, then you can give the command **:sh**CR. This will give you a new shell, and when you finish with the shell, ending it by typing a **^D**, the editor will clear the screen and continue.

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

5.3 Marking and Returning

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

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

5.4 Adjusting the Screen

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

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

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

6. Special Topics

6.1 Editing on Slow Terminals

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

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

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

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

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

```
": / ? [[ ]] ` ´"
```

Thus if you are searching for a particular instance of a common string

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

You can easily expand or contract the window, placing the current line as you choose, by giving a number on a **z** command, after the **z** and before the following RETURN, **.** or **–**. Thus the command **z5.** redraws the screen with the current line in the center of a five line window.[16]

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

See section 8.8 on *open* mode for another way to use the *vi* command set on slow terminals.

6.2 Options, Set, and Editor Startup Files

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

16. Note that the command **5z.** has an entirely different effect, placing line 5 in the center of a new window.

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

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

set *opt=val*

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

set *opt*
set no*opt*

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

You can get a list of all options which you have changed by the command **:set**CR, or the value of a single option by the command **:set** *opt***?**CR. A list of all possible options and their values is generated by

:set allCR. Set can be abbreviated **se**. Multiple options can be placed on one line, e.g. **:se ai aw nu**CR.

Options set by the **set** command only last while you stay in the editor. It is common to want to have certain options set whenever you use the editor. This can be accomplished by creating a list of *ex* commands[17] which are to be run every time you start up *ex*, *edit*, or *vi*. A typical list includes a **set** command, and possibly a few **map** commands. Since it is advisable to get these commands on one line, they can be separated with the | character, for example:

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

which sets the options *autoindent*, *autowrite*, *terse*, (the **set** command), makes @ delete a line, (the first **map**), and makes # delete a character, (the second **map**). (See section 6.9 for a description of the **map** command) This string should be placed in the variable EXINIT in your environment. If you use the shell *csh*, put this line in the file *.login* in your home directory:

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

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

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

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

6.3 Recovering Lost Lines

You might have a serious problem if you delete a number of lines and then regret that they were deleted. Despair not, the editor saves the last 9 deleted blocks of text in a set of numbered registers 1–9. You can get the *n*'th previous deleted text back in your file by the command "*n***p**. The " here says that a buffer name is to follow, *n* is the number of the buffer you wish to try (use the number 1 for now), and **p** is the put command, which puts text in the buffer after the cursor. If this doesn't bring back the text you wanted, hit **u** to undo this

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

and then . (period) to repeat the put command. In general the . command will repeat the last change you made. As a special case, when the last command refers to a numbered text buffer, the . command increments the number of the buffer before repeating the command. Thus a sequence of the form

"1pu.u.u.

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

6.4 Recovering Lost Files

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

% **vi –r** *name*

replacing *name* with the name of the file which you were editing. This will recover your work to a point near where you left off.[18]

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

% **vi –r**

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

For this feature to work, *vi* must be correctly installed by a super user on your system, and the *mail* program must exist to receive mail.

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

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

6.5 Continuous Text Input

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

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

6.6 Features for Editing Programs

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

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

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

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

and >L shifting the rest of the display left and right.

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

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

6.7 Filtering Portions of the Buffer

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

6.8 Commands for Editing LISP

If you are editing a LISP program you should set the option *lisp* by doing **:se lisp**CR. This changes the (and) commands to move backward and forward over s-expressions. The { and } commands are like (and) but don't stop at atoms. These can be used to skip to the next list, or through a comment quickly.

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

There is another option which is useful for typing in LISP, the *showmatch* option. Try setting it with **:se sm**CR and then try typing a '(' some words and then a ')'. Notice that the cursor shows the position of the '(' which matches the ')' briefly. This happens only if the matching '(' is on the screen, and the cursor stays there for at most one second.

The editor also has an operator to realign existing lines as though they had been typed in with *lisp* and *autoindent* set. This is the =

operator. Try the command =% at the beginning of a function. This will realign all the lines of the function declaration.

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

6.9 Macros

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

Briefly, there are two flavors of macros:

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

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

:map *lhs rhs*CR

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

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

:map q :wq^**V**^**V**CR CR

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

Macros can be deleted with

unmap lhs

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

:map ^V^V^I #

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

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

Placing a '!' after the word **map** causes the mapping to apply to input mode, rather than command mode. Thus, to arrange for **^T** to be the same as 4 spaces in input mode, you can type:

:map ^T ^V␣␣␣␣

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

7. Word Abbreviations

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

:ab eecs Electrical Engineering and Computer Sciences

causes the word "eecs" to always be changed into the phrase "Electrical Engineering and Computer Sciences". Word abbreviation is different from macros in that only whole words are affected. If "eecs" were typed as part of a larger word, it would be left alone. Also, the partial word is echoed as it is typed. There is no need for an abbreviation to be a single keystroke, as it should be with a macro.

7.1 Abbreviations

The editor has a number of short commands which abbreviate longer commands which we have introduced here. You can find these commands easily on the quick reference card. They often save a bit of typing and you can learn them as convenient.

8. Nitty-gritty Details

8.1 Line Representation in the Display

The editor folds long logical lines onto many physical lines in the display. Commands which advance lines advance logical lines and will skip over all the segments of a line in one motion. The command | moves the cursor to a specific column, and may be useful for getting near the middle of a long line to split it in half. Try **80|** on a line which is more than 80 columns long.[19]

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

If you wish, you can have the editor place line numbers before each line on the display. Give the command **:se nu**CR to enable this, and the command **:se nonu**CR to turn it off. You can have tabs represented as **^I** and the ends of lines indicated with '$' by giving the command **:se list**CR; **:se nolist**CR turns this off.

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

19. You can make long lines very easily by using **J** to join together short lines.

8.2 Counts

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

new window size	**: / ? [[ ]] ` '**
scroll amount	**^D ^U**
line/column number	**z G |**
repeat effect	most of the rest

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

This size is the size used when the editor clears and refills the screen after a search or other motion moves far from the edge of the current window. The commands which take a new window size as count all often cause the screen to be redrawn. If you anticipate this, but do not need as large a window as you are currently using, you may wish to change the screen size by specifying the new size before these commands. In any case, the number of lines used on the screen will expand if you move off the top with a **–** or similar command or off the bottom with a command such as RETURN or **^D**. The window will revert to the last specified size the next time it is cleared and refilled.[20]

The scroll commands **^D** and **^U** likewise remember the amount of scroll last specified, using half the basic window size initially. The simple insert commands use a count to specify a repetition of the inserted text. Thus **80a+----**ESC will insert a grid-like string of text. A few commands also use a preceding count as a line or column number.

Except for a few commands which ignore any counts (such as **^R**), the rest of the editor commands use a count to indicate a simple repetition of their effect. Thus **5w** advances five words on the current line, while **5**RETURN advances five lines. A very useful instance of a count as a repetition is a count given to the **.** command,

20. But not by a **^L** which just redraws the screen as it is.

which repeats the last changing command. If you do **dw** and then **3.**, you will delete first one and then three words. You can then delete two more words with **2.**.

8.3 More File Manipulation Commands

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

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

All of these commands are followed by a CR or ESC. The most basic commands are **:w** and **:e**. A normal editing session on a single file will end with a **ZZ** command. If you are editing for a long period of time you can give **:w** commands occasionally after major amounts of editing, and then be sure you end with a **ZZ**. When you edit more than one file, you can finish editing one file with a **:w** and start editing a new file by giving a **:e** command, or set *autowrite* and use **:n** <file>.

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

The **:e** command can be given a **+** argument to start at the end of the file, or a **+**n argument to start at line n. In actuality, n may be any editor command not containing a space, usefully a scan like **+**/*pat* or

+?*pat*. In forming new names to the **e** command, you can use the character **%** which is replaced by the current file name, or the character **#** which is replaced by the alternate file name. The alternate file name is generally the last name you typed other than the current file. Thus if you try to do a **:e** and get a diagnostic that you haven't written the file, you can give a **:w** command and then a **:e #** command to redo the previous **:e**.

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

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

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

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

8.4 More About Searching for Strings

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

You can have the editor ignore the case of words in the searches it does by giving the command **:se ic**CR. The command **:se noic**CR turns this off.

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

```
set nomagic
```

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

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

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

8.5 More About Input Mode

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

^H	deletes the last input character
^W	deletes the last input word, defined as by **b**
erase	your erase character, same as **^H**
kill	your kill character, deletes the input on this line
\	escapes a following **^H** and your erase and kill
ESC	ends an insertion
DEL	interrupts an insertion, terminating it abnormally
CR	starts a new line
^D	backtabs over *autoindent*

0^D	kills all the *autoindent*
↑^D	same as **0^D**, but restores indent next line
^V	quotes the next non-printing character into the file

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

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

If you wish to type in your erase or kill character (say # or @) then you must precede it with a \, just as you would do at the normal system command level. A more general way of typing non-printing characters into the file is to precede them with a **^V**. The **^V** echoes as a ↑ character on which the cursor rests. This indicates that the editor expects you to type a control character. In fact you may type any character and it will be inserted into the file at that point.[21]

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

When you are using *autoindent* you may wish to place a label at the left margin of a line. The way to do this easily is to type ↑ and then **^D**. The editor will move the cursor to the left margin for one line,

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

and restore the previous indent on the next. You can also type a **0** followed immediately by a **^D** if you wish to kill all the indent and not have it come back on the next line.

8.6 Upper Case Only Terminals

If your terminal has only upper case, you can still use *vi* by using the normal system convention for typing on such a terminal. Characters which you normally type are converted to lower case, and you can type upper case letters by preceding them with a \. The characters { ~ } | ` are not available on such terminals, but you can escape them as \(\^ \) \! \'. These characters are represented on the display in the same way they are typed.[22]

8.7 Vi and Ex

Vi is actually one mode of editing within the editor *ex*. When you are running *vi* you can escape to the line oriented editor of *ex* by giving the command **Q**. All of the **:** commands which were introduced above are available in *ex*. Likewise, most *ex* commands can be invoked from *vi* using **:**. Just give them without the **:** and follow them with a CR.

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

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

22. The \ character you give will not echo until you type another key.

8.8 Open Mode: Vi on Hardcopy Terminals and "glass tty's"

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

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

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

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

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

Acknowledgements

Bruce Englar encouraged the early development of this display editor. Peter Kessler helped bring sanity to version 2's command layout. Bill Joy wrote versions 1 and 2.0 through 2.7, and created the framework that users see in the present editor. Mark Horton added macros and other features and made the editor work on a large number of terminals and Unix systems.

Appendix: Character Functions

This appendix gives the uses the editor makes of each character. The characters are presented in their order in the ASCII character set: Control characters come first, then most special characters, then the digits, upper and then lower case characters.

For each character we tell a meaning it has as a command and any meaning it has during an insert. If it has only meaning as a command, then only this is discussed. Section numbers in parentheses indicate where the character is discussed; a 'f' after the section number means that the character is mentioned in a footnote.

^@ Not a command character. If typed as the first character of an insertion it is replaced with the last text inserted, and the insert terminates. Only 128 characters are saved from the last insert; if more characters were inserted the mechanism is not available. A **^@** cannot be part of the file due to the editor implementation (8.5f).

^A Unused.

^B Backward window. A count specifies repetition. Two lines of continuity are kept if possible (2.1, 7.1, 8.2).

^C Unused.

^D As a command, scrolls down a half-window of text. A count gives the number of (logical) lines to scroll, and is remembered for future **^D** and **^U** commands (2.1, 8.2). During an insert, backtabs over *autoindent* white space at the beginning of a line (6.6, 8.5); this white space cannot be backspaced over.

^E Exposes one more line below the current screen in the file, leaving the cursor where it is if possible. (Version 3 only.)

^F Forward window. A count specifies repetition. Two lines of continuity are kept if possible (2.1, 7.1, 8.2).

^G Equivalent to **:f**CR, printing the current file, whether it has been modified, the current line number and the number of lines in the file, and the percentage of the way through the file that you are.

^H (BS) Same as **left arrow**. (See **h**). During an insert, eliminates the last input character, backing over it but not erasing it;

it remains so you can see what you typed if you wish to type something only slightly different (2.4, 8.5).

^I(TAB) Not a command character. When inserted it prints as some number of spaces. When the cursor is at a tab character it rests at the last of the spaces which represent the tab. The spacing of tabstops is controlled by the *tabstop* option (4.1, 6.2).

^J(LF) Same as **down arrow** (see **j**).

^K Unused.

^L The ASCII formfeed character, this causes the screen to be cleared and redrawn. This is useful after a transmission error, if characters typed by a program other than the editor scramble the screen, or after output is stopped by an interrupt (5.4, 8.2f).

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

^N Same as **down arrow** (see **j**).

^O Unused.

^P Same as **up arrow** (see **k**).

^Q Not a command character. In input mode, ^Q quotes the next character, the same as ^V, except that some teletype drivers will eat the ^Q so that the editor never sees it.

^R Redraws the current screen, eliminating logical lines not corresponding to physical lines (lines with only a single @ character on them). On hardcopy terminals in *open* mode, retypes the current line (5.4, 8.1, 8.8).

^S Unused. Some teletype drivers use ^S to suspend output until ^Q is pressed.

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

^U Scrolls the screen up, inverting ^D which scrolls down. Counts work as they do for ^D, and the previous scroll amount is common to both. On a dumb terminal, ^U will often necessitate clearing and redrawing the screen further back in the file (2.1, 8.2).

^V Not a command character. In input mode, quotes the next character so that it is possible to insert non-printing and special characters into the file (6.9, 8.5).

^W Not a command character. During an insert, backs up as **b** would in command mode; the deleted characters remain on the display (see ^H) (8.5).

^X Unused.

^Y Exposes one more line above the current screen, leaving the cursor where it is if possible. (No mnemonic value for this key; however, it is next to ^U which scrolls up a bunch.) (Version 3 only.)

^Z If supported by the Unix system, stops the editor, exiting to the top level shell. Same as **:stop**CR. Otherwise, unused.

^[(ESC) Cancels a partially formed command, such as a **z** when no following character has yet been given; terminates inputs on the last line (read by commands such as **:** / and **?**); ends insertions of new text into the buffer. If an ESC is given when quiescent in command state, the editor rings the bell or flashes the screen. You can thus hit ESC if you don't know what is happening till the editor rings the bell. If you don't know if you are in insert mode you can type ESC**a**, and then material to be input; the material will be inserted correctly whether or not you were in insert mode when you started (1.6, 3.1, 8.5).

^ | \ Unused.

^] Searches for the word which is after the cursor as a tag. Equivalent to typing **:ta**, this word, and then a CR. Mnemonically, this command is "go right to" (8.3).

^↑ Equivalent to **:e #**CR, returning to the previous position in the last edited file, or editing a file which you specified if you got a "No write since last change diagnostic" and do not want to have to type the file name again (8.3). (You have to do a **:w** before ^↑ will work in this case. If you do not wish to write the file you should do **:e! #**CR instead.)

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

SPACE Same as **right arrow** (see **l**).

! An operator, which processes lines from the buffer with reformatting commands. Follow **!** with the object to be processed, and then the command name terminated by CR. Doubling **!** and preceding it by a count causes count lines to be filtered; otherwise the count is passed on to the object after the **!**. Thus **2!}** *fmt*CR reformats the next two paragraphs by running them through the program *fmt*. If you are working on LISP, the command **!%***grind*CR,[23] given at the beginning of a function, will run the text of the function through the LISP grinder (6.7, 8.3). To read a file or the output of a command into the buffer use **:r** (8.3). To simply execute a command use **:!** (8.3).

" Precedes a named buffer specification. There are named buffers **1–9** used for saving deleted text and named buffers **a–z** into which you can place text (4.3, 6.3)

The macro character which, when followed by a number, will substitute for a function key on terminals without function keys (6.9). In input mode, if this is your erase character, it will delete the last character you typed in input mode, and must be preceded with a \ to insert it, since it normally backs over the last input character you gave.

$** Moves to the end of the current line. If you **:se list**CR, then the end of each line will be shown by printing a **$ after the end of the displayed text in the line. Given a count, advances to the count'th following end of line; thus **2$** advances to the end of the following line.

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

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

' When followed by a **'** returns to the previous context at the beginning of a line. The previous context is set whenever the current line is moved in a non-relative way.

23. Both *fmt* and *grind* are Berkeley programs and may not be present at all installations.

When followed by a letter **a–z**, returns to the line which was marked with this letter with a **m** command, at the first non-white character in the line. (2.2, 5.3). When used with an operator such as **d**, the operation takes place over complete lines; if you use `, the operation takes place from the exact marked place to the current cursor position within the line.

(Retreats to the beginning of a sentence, or to the beginning of a LISP s-expression if the *lisp* option is set. A sentence ends at a . ! or ? which is followed by either the end of a line or by two spaces. Any number of closing)] " and ' characters may appear after the . ! or ?, and before the spaces or end of line. Sentences also begin at paragraph and section boundaries (see { and [[below). A count advances that many sentences (4.2, 6.8).

) Advances to the beginning of a sentence. A count repeats the effect. See (above for the definition of a sentence (4.2, 6.8).

* Unused.

+ Same as CR when used as a command.

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

– Retreats to the previous line at the first non-white character. This is the inverse of + and RETURN. If the line moved to is not on the screen, the screen is scrolled, or cleared and redrawn if this is not possible. If a large amount of scrolling would be required the screen is also cleared and redrawn, with the current line at the center (2.3).

. Repeats the last command which changed the buffer. Especially useful when deleting words or lines; you can delete some words/lines and then hit . to delete more and more words/lines. Given a count, it passes it on to the command being repeated. Thus after a **2dw**, **3.** deletes three words (3.3, 6.3, 8.2, 8.4).

/ Reads a string from the last line on the screen, and scans forward for the next occurrence of this string. The normal input editing sequences may be used during the input on the bottom line; an returns to command state without

ever searching. The search begins when you hit CR to terminate the pattern; the cursor moves to the beginning of the last line to indicate that the search is in progress; the search may then be terminated with a DEL or RUB, or by backspacing when at the beginning of the bottom line, returning the cursor to its initial position. Searches normally wrap end-around to find a string anywhere in the buffer.

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

To include the character / in the search string, you must escape it with a preceding \. A ↑ at the beginning of the pattern forces the match to occur at the beginning of a line only; this speeds the search. A **$** at the end of the pattern forces the match to occur at the end of a line only. More extended pattern matching is available, see section 7.4; unless you set **nomagic** in your *.exrc* file you will have to preceed the characters **.** **[** ***** and **~** in the search pattern with a \ to get them to work as you would naively expect (1.6, 2.2, 6.1, 8.2, 8.4).

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

1–9 Used to form numeric arguments to commands (2.3, 8.2).

: A prefix to a set of commands for file and option manipulation and escapes to the system. Input is given on the bottom line and terminated with an CR, and the command then executed. You can return to where you were by hitting DEL or RUB if you hit **:** accidentally (see primarily 6.2 and 8.3).

; Repeats the last single character find which used **f** **F** **t** or **T**. A count iterates the basic scan (4.1).

< An operator which shifts lines left one *shiftwidth*, normally 8 spaces. Like all operators, affects lines when repeated, as in **<<**. Counts are passed through to the basic object, thus **3<<** shifts three lines (6.6, 8.2).

= Reindents line for LISP, as though they were typed in with *lisp* and *autoindent* set (6.8).

> An operator which shifts lines right one *shiftwidth*, normally 8 spaces. Affects lines when repeated as in >>. Counts repeat the basic object (6.6, 8.2).

? Scans backwards, the opposite of /. See the / description above for details on scanning (2.2, 6.1, 8.4).

@ A macro character (6.9). If this is your kill character, you must escape it with a \ to type it in during input mode, as it normally backs over the input you have given on the current line (3.1, 3.4, 8.5).

A Appends at the end of line, a synonym for **$a** (8.2).

B Backs up a word, where words are composed of non-blank sequences, placing the cursor at the beginning of the word. A count repeats the effect (2.4).

C Changes the rest of the text on the current line; a synonym for **c$**.

D Deletes the rest of the text on the current line; a synonym for **d$**.

E Moves forward to the end of a word, defined as blanks and non-blanks, like **B** and **W**. A count repeats the effect.

F Finds a single following character, backwards in the current line. A count repeats this search that many times (4.1).

G Goes to the line number given as preceding argument, or the end of the file if no preceding count is given. The screen is redrawn with the new current line in the center if necessary (2.2).

H **Home arrow**. Homes the cursor to the top line on the screen. If a count is given, then the cursor is moved to the count'th line on the screen. In any case the cursor is moved to the first non-white character on the line. If used as the target of an operator, full lines are affected (2.3, 3.2).

I Inserts at the beginning of a line; a synonym for **^i**.

J Joins together lines, supplying appropriate white space: one space between words, two spaces after a ., and no

spaces at all if the first character of the joined on line is). A count causes that many lines to be joined rather than the default two (6.5, 8.1f).

K Unused.

L Moves the cursor to the first non-white character of the last line on the screen. With a count, to the first non-white of the count'th line from the bottom. Operators affect whole lines when used with **L** (2.3).

M Moves the cursor to the middle line on the screen, at the first non-white position on the line (2.3).

N Scans for the next match of the last pattern given to / or **?**, but in the reverse direction; this is the reverse of **n**.

O Opens a new line above the current line and inputs text there up to an ESC. A count can be used on dumb terminals to specify a number of lines to be opened; this is generally obsolete, as the *slowopen* option works better (3.1).

P Puts the last deleted text back before/above the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines. Otherwise the text is inserted between the characters before and at the cursor. May be preceded by a named buffer specification "*x* to retrieve the contents of the buffer; buffers **1–9** contain deleted material, buffers **a–z** are available for general use (4.3).

Q Quits from *vi* to *ex* command mode. In this mode, whole lines form commands, ending with a RETURN. You can give all the **:** commands; the editor supplies the **:** as a prompt (8.7).

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

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

T Takes a single following character, locates the character before the cursor in the current line, and places the cursor just after that character. A count repeats the effect. Most useful with operators such as **d** (4.1).

U Restores the current line to its state before you started changing it (3.5).

V Unused.

W Moves forward to the beginning of a word in the current line, where words are defined as sequences of blank/non-blank characters. A count repeats the effect (2.4).

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

Y Yanks a copy of the current line into the unnamed buffer, to be put back by a later **p** or **P**; a very useful synonym for **yy**. A count yanks that many lines. May be preceded by a buffer name to put lines in that buffer (4.3).

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

[[Backs up to the previous section boundary. A section begins at each macro in the *sections* option, normally a '.NH' or '.SH' and also at lines which which start with a formfeed **^L**. Lines beginning with **{** also stop **[[**; this makes it useful for looking backwards, a function at a time, in C programs. If the option *lisp* is set, stops at each (at the beginning of a line, and is thus useful for moving backwards at the top level LISP objects. (4.2, 6.1, 6.6, 8.2).

**** Unused.

]] Forward to a section boundary, see **[[** for a definition (4.2, 6.1, 6.6, 8.2).

↑ Moves to the first non-white position on the current line (4.4).

_ Unused.

`** When followed by a **` returns to the previous context. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter **a–z**, returns to the position which was marked with this letter with a **m** command. When used with an operator such as **d**, the operation takes place from the exact marked place to the current position within the line; if you use **´**, the operation takes place over complete lines (2.2, 5.3).

a Appends arbitrary text after the current cursor position; the insert can continue onto multiple lines by using RETURN within the insert. A count causes the inserted text to be replicated, but only if the inserted text is all on one line. The insertion terminates with an ESC (3.1, 8.2).

b Backs up to the beginning of a word in the current line. A word is a sequence of alphanumerics, or a sequence of special characters. A count repeats the effect (2.4).

c An operator which changes the following object, replacing it with the following input text up to an ESC. If more than part of a single line is affected, the text which is changed away is saved in the numeric named buffers. If only part of the current line is affected, then the last character to be changed away is marked with a **$**. A count causes that many objects to be affected, thus both **3c)** and **c3)** change the following three sentences (8.4).

d An operator which deletes the following object. If more than part of a line is affected, the text is saved in the numeric buffers. A count causes that many objects to be affected; thus **3dw** is the same as **d3w** (3.3, 3.4, 4.1, 8.4).

e Advances to the end of the next word, defined as for **b** and **w**. A count repeats the effect (2.4, 3.1).

f Finds the first instance of the next character following the cursor on the current line. A count repeats the find (4.1).

g Unused.

h **Left arrow**. Moves the cursor one character to the left. Like the other arrow keys, either **h**, the **left arrow** key, or one of the synonyms (**^H**) has the same effect. On v2 editors, arrow keys on certain kinds of terminals (those which send escape sequences, such as vt52, c100, or hp) cannot be used. A count repeats the effect (3.1, 8.5).

i Inserts text before the cursor, otherwise like **a** (8.2).

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

k **Up arrow**. Moves the cursor one line up. **^P** is a synonym.

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

m Marks the current position of the cursor in the mark register which is specified by the next character **a–z**. Return to this position or use with an operator using ` or ' (5.3).

n Repeats the last / or **?** scanning commands (2.2).

o Opens new lines below the current line; otherwise like **O** (3.1).

p Puts text after/below the cursor; otherwise like **P** (6.3).

q Unused.

r Replaces the single character at the cursor with a single character you type. The new character may be a RETURN; this is the easiest way to split lines. A count replaces each of the following count characters with the single character given; see **R** above which is the more usually useful iteration of **r** (3.2).

s Changes the single character under the cursor to the text which follows up to an ESC; given a count, that many characters from the current line are changed. The last character to be changed is marked with **$** as in **c** (3.2).

t Advances the cursor upto the character before the next character typed. Most useful with operators such as **d** and **c** to delete the characters up to a following character. You can use **.** to delete more if this doesn't delete enough the first time (4.1).

u Undoes the last change made to the current buffer. If repeated, will alternate between these two states, thus is its own inverse. When used after an insert which inserted text on more than one line, the lines are saved in the numeric named buffers (3.5).

v Unused.

w Advances to the beginning of the next word, as defined by **b** (2.4).

x Deletes the single character under the cursor. With a count deletes deletes that many characters forward from the cursor position, but only on the current line (6.5).

y An operator, yanks the following object into the unnamed temporary buffer. If preceded by a named buffer specification, "*x*, the text is placed in that buffer also. Text can be recovered by a later **p** or **P** (8.4).

z Redraws the screen with the current line placed as specified by the following character: RETURN specifies the top of the screen, . the center of the screen, and – at the bottom of the screen. A count may be given after the **z** and before the following character to specify the new screen size for the redraw. A count before the **z** gives the number of the line to place in the center of the screen instead of the default current line. (5.4)

{ Retreats to the beginning of the beginning of the preceding paragraph. A paragraph begins at each macro in the *paragraphs* option, normally '.IP', '.LP', '.PP', '.QP' and '.bp'. A paragraph also begins after a completely empty line, and at each section boundary (see **[[** above) (4.2, 6.8, 8.6).

| Places the cursor on the character in the column specified by the count (8.1, 8.2).

} Advances to the beginning of the next paragraph. See **{** for the definition of paragraph (4.2, 6.8, 8.6).

~ Unused.

^?L) Interrupts the editor, returning it to command accepting state (1.6, 8.5)

5

Vi Command & Function Reference

1. Author's Disclaimer

This document does not claim to be 100% complete. There are a few commands listed in the original document that I was unable to test either because I do not speak **lisp**, because they required programs we don't have, or because I wasn't able to make them work. In these cases I left the command out. The commands listed in this document have been tried and are known to work. It is expected that prospective users of this document will read it once to get the flavor of everything that **vi** can do and then use it as a reference document. Experimentation is recommended. If you don't understand a command, try it and see what happens.

NOTE

In revising this document, I have attempted to make it completely reflect version 2.12 of **vi**. It does not attempt to document the VAX version (version 3), but with one or two exceptions (wrapmargin, arrow keys) everything said about 2.12 should apply to 3.1. *Mark Horton*

Source: Alan P.W. Hewett (Revised for version 2.12 by Mark Horton), *Vi Command & Function Reference* (Berkeley, CA: University of California).

2. Notation

[option] is used to denote optional parts of a command. Many **vi** commands have an optional count. **[#]** means that an optional number may precede the command to multiply or iterate the command. **{variable item}** is used to denote parts of the command which must appear, but can take a number of different values. **<character [-character]>** means that the character or one of the characters in the range described between the two angle brackets is to be typed. For example **<esc>** means the **escape** key is to be typed. **<a-z>** means that a lower case letter is to be typed. **^<character>** means that the character is to be typed as a **control** character, that is, with the **<#l>** key held down while simultaneously typing the specified character. In this document control characters will be denoted using the *upper case* character, but ^<uppercase chr> and ^<lowercase chr> are equivalent. That is, for example, **<^D>** is equal to **<^d>**. The most common character abbreviations used in this list are as follows:

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

3. Basics

To run **vi** the shell variable **TERM** must be defined and exported to your environment. How you do this depends on which shell you are using. You can tell which shell you have by the character it prompts you for commands with. The Bourne shell prompts with '$', and the

C shell prompts with '%'. For these examples, we will suppose that you are using an HP 2621 terminal, whose termcap name is "2621".

3.1 Bourne Shell

To manually set your terminal type to 2621 you would type:

```
TERM=2621
export TERM
```

There are various ways of having this automatically or semi-automatically done when you log in. Suppose you usually dial in on a 2621. You want to tell this to the machine, but still have it work when you use a hardwired terminal. The recommended way, if you have the **tset** program, is to use the sequence

```
tset -s -d 2621 > tset$$
. tset$$
rm tset$$
```

in your .login (for csh) or the same thing using '.' instead of 'source' in your .profile (for sh). The above line says that if you are dialing in you are on a 2621, but if you are on a hardwired terminal it figures out your terminal type from an on-line list.

3.2 The C Shell

To manually set your terminal type to 2621 you would type:

```
setenv TERM 2621
```

There are various ways of having this automatically or semi-automatically done when you log in. Suppose you usually dial in on a 2621. You want to tell this to the machine, but still have it work when you use a hardwired terminal. The recommended way, if you have the **tset** program, is to use the sequence

```
tset -s -d 2621 > tset$$
source tset$$
rm tset$$
```

in your .login.* The above line says that if you are dialing in you are

on a 2621, but if you are on a hardwired terminal it figures out your terminal type from an on-line list.

4. Normal Commands

Vi is a visual editor with a window on the file. What you see on the screen is **vi**'s current notion of what your file will contain, (at this point in the file), when it is written out. Most commands do not cause any change in the screen until the complete command is typed. Should you get confused while typing a command, you can abort the command by typing an <del> character. You will know you are back to command level when you hear a <bell>. Usually typing an <esc> will produce the same result. When **vi** gets an improperly formatted command it rings the <bell>. Following are the **vi** commands broken down by function.

4.1 Entry and Exit

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

vi *file*

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

To get out of the editor, type

ZZ

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

---

* On a version 6 system without environments, the invocation of tset is simpler, just add the line "tset –d 2621" to your .login or .profile.

4.2 Cursor and Page Motion

NOTE

The arrow keys (see the next four commands) on certain kinds of terminals will not work with the PDP-11 version of vi. The control versions or the hjkl versions will work on any terminal. Experienced users prefer the hjkl keys because they are always right under their fingers. Beginners often prefer the arrow keys, since they do not require memorization of which hjkl key is which. The mnemonic value of hjkl is clear from looking at the keyboard of an adm3a.

[#]<bs> or [#]h or [#]←	Move the cursor to the left one character. Cursor stops at the left margin of the page. If # is given, these commands move that many spaces.
[#]^N or [#]j or [#]↓ or [#]<lf>	Move down one line. Moving off the screen scrolls the window to force a new line onto the screen. Mnemonic: Next
[#]^P or [#]k or [#]↑	Move up one line. Moving off the top of the screen forces new text onto the screen. Mnemonic: Previous
[#]<sp> or [#]l or [#]→	Move to the right one character. Cursor will not go beyond the end of the line.
[#]-	Move the cursor up the screen to the beginning of the next line. Scroll if necessary.
[#]+	Move the cursor down the screen to the beginning of the next line. Scroll up if necessary.
[#]$	Move the cursor to the end of the line. If there is a count, move to the end of the line "#" lines forward in the file.
^	Move the cursor to the beginning of the first word on the line.
0	Move the cursor to the left margin of the current line.
[#]|	Move the cursor to the column specified by the count. The default is column zero.

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

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

[#]b — Move the cursor to the preceding word. Mnemonic: backup-word

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

[#]e — Move the cursor to the end of the current word or the end of the "#"th word hence. Mnemonic: end-of-word

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

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

NOTE

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

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

[#]^U — Move the cursor up in the file by "#" lines. The screen is simultaneously scrolled down. Mnemonic: Up

[#]^F — Move the cursor to the next page. A count moves that many pages. Two lines of the previous page

are kept on the screen for continuity if possible. Mnemonic: Forward-a-page

[#]^B — Move the cursor to the previous page. Two lines of the current page are kept if possible. Mnemonic: Backup-a-page

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

[#]) — Move the cursor backwards to the beginning of a sentence.

[#]} — Move the cursor to the beginning of the next paragraph. This command works best inside **nroff** documents. It understands two sets of **nroff** macros, **–ms** and **–mm**, for which the commands ".IP", ".LP", ".PP", ".QP", ".P", as well as the nroff command ".bp" are considered to be paragraph delimiters. A blank line also delimits a paragraph. The **nroff** macros that it accepts as paragraph delimiters is adjustable. See **paragraphs** under the **Set Commands** section.

[#]{ — Move the cursor backwards to the beginning of a paragraph.

]] — Move the cursor to the next "section", where a section is defined by two sets of **nroff** macros, **–ms** and **–mm**, in which ".NH", ".SH", and ".H" delimit a section. A line beginning with a <ff><nl> sequence, or a line beginning with a "{" are also considered to be section delimiters. The last option makes it useful for finding the beginnings of C functions. The **nroff** macros that are used for section delimiters can be adjusted. See **sections** under the **Set Commands** section.

[[— Move the cursor backwards to the beginning of a section.

% — Move the cursor to the matching parenthesis or brace. This is very useful in C or lisp code. If the cursor is sitting on a ( ) { or } the cursor is moved to the matching character at the other end of the section. If the cursor is not sitting on a brace or a

parenthesis, **vi** searches forward until it finds one and then jumps to the match mate.

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

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

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

m<a-z> — This command does not move the cursor, but it **marks** the place in the file and the character "<a-z>" becomes the label for referring to this location in the file. See the next two commands. Mnemonic: **m**ark

NOTE

> The mark command is not a motion, and cannot be used as the target of commands such as delete.

'<a-z> — Move the cursor to the beginning of the line that is marked with the label "<a-z>".

`<a-z> — Move the cursor to the exact position on the line that was marked with with the label "<a-z>".

'' — Move the cursor back to the beginning of the line where it was before the last "non-relative" move. A "non-relative" move is something such as a search or a jump to a specific line in the file, rather than moving the cursor or scrolling the screen.

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

4.3 Searches

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

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

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

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

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

[#]; Repeat the last "f", "F", "t" or "T" command.

[#], Repeat the last "f", "F", "t" or "T" command, but in the opposite search direction. This is useful if you overshoot.

[#]/[string]/<nl>

Search forward for the next occurrence of "string". Wrap around at the end of the file does occur. The final </> is not required.

[#]?[string]?<nl>

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

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

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

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

Using the : syntax it is possible to do global searches ala the standard UNIX "ed" editor.

4.4 Text Insertion

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

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

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

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

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

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

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

4.5 Text Deletion

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

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

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

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

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

4.6 Text Replacement

The following commands allow the user to simultaneously delete and insert new text. All such actions can be **undone** by typing **u** following the command.

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

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

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

[#]S{text}<esc>

Substitute for the entire current line (or lines). If no count is given, a "$" appears at the end of the current line. If a count of more than 1 is given, all the lines to be replaced are deleted before the insertion begins.

[#]c{motion}{text}<esc>

Change the specified "motion" by replacing it with the insertion text. A "$" will appear at the end of the last item that is being deleted unless the deletion involves whole lines. Motion's can be any motion from sections 4.1 or 4.2. Stuttering the c (e.g. [#]cc) changes # lines.

4.7 Moving Text

Vi provides a number of ways of moving chunks of text around. There are nine buffers into which each piece of text which is deleted or "yanked" is put in addition to the "undo" buffer. The most recent deletion or yank is in the "undo" buffer and also usually in buffer 1, the next most recent in buffer 2, and so forth. Each new

deletion pushes down all the older deletions. Deletions older than 9 disappear. There is also a set of named registers, a-z, into which text can optionally be placed. If any delete or replacement type command is preceded by **"<a-z>**, that named buffer will contain the text deleted after the command is executed. For example, **"a3dd** will delete three lines starting at the current line and put them in buffer **"a.*** There are two more basic commands and some variations useful in getting and putting text into a file.

["<a-z>][#]y{motion}

Yank the specified item or "#" items and put in the "undo" buffer or the specified buffer. The variety of "items" that can be yanked is the same as those that can be deleted with the "d" command or changed with the "c" command. In the same way that "dd" means delete the current line and "cc" means replace the current line, "yy" means yank the current line.

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

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

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

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

lost when changing files.

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

[#]>{motion} — The shift operator will right shift all the text from the line on which the cursor is located to the line where the **motion** is located. The text is shifted by one **shiftwidth**. (See section 6.) >> means right shift the current line or lines.

[#]<{motion} — The shift operator will left shift all the text from the line on which the cursor is located to the line where the **item** is located. The text is shifted by one **shiftwidth**. (See section 6.) << means left shift the current line or lines. Once the line has reached the left margin it is not further affected.

[#]={motion} — Prettyprints the indicated area according to **lisp** conventions. The area should be a lisp s-expression.

4.8 Miscellaneous Commands

Vi has a number of miscellaneous commands that are very useful. They are:

ZZ — This is the normal way to exit from vi. If any changes have been made, the file is written out. Then you are returned to the shell.

^L — Redraw the current screen. This is useful if someone "write"s you while you are in "vi" or if for any reason garbage gets onto the screen.

^R — On dumb terminals, those not having the "delete line" function (the vt100 is such a terminal), **vi** saves redrawing the screen when you delete a line by just marking the line with an "@" at the beginning and blanking the line. If you want to actually get rid of the lines marked with "@" and see what the page looks like, typing a ^R will do this.

. "Dot" is a particularly useful command. It repeats the last text modifying command. Therefore you can type a command once and then to another place and repeat it by just typing ".".

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

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

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

Q Switch to **ex** editing mode. In this mode **vi** will behave very much like **ed**. The editor in this mode will operate on single lines normally and will not attempt to keep the "window" up to date. Once in this mode it is also possible to switch to the **open** mode of editing. By entering the command **[line number]open<nl>** you enter this mode. It is similar to the normal visual mode except the window is only **one** line long. Mnemonic: **Q**uit visual mode

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

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

Any UNIX filter (e.g. command that reads the standard input and outputs something to the standard output) can be sent a section of the current file and have the output of the command replace the original text. Useful examples are programs like **cb**, **sort**, and **nroff**. For instance, using **sort** it would be possible to sort a section of the current file into a new list. Using **!!** means take a line or lines starting at the line the cursor is currently on and pass them to the UNIX command.

NOTE

To just escape to the shell for one command, use :![cmd]<nl>, see section 5.

z[#]<nl>

This resets the current window size to "#" lines and redraws the screen.

4.9 Special Insert Characters

There are some characters that have special meanings during insert modes. They are:

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

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

^W — If the cursor is sitting on a word, <^W> moves the cursor back to the beginning of the word, thus erasing the word from the insert. Mnemonic: erase Word

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

5. : Commands

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

:w[!] [file] Causes **vi** to write out the current text to the disk. It is written to the file you are editing unless "file" is supplied. If "file" is supplied, the write is directed to that file instead. If that file already exists, **vi** will not perform the write unless the "!" is supplied indicating you *really* want to destroy the older copy of the file.

:q[!] Causes **vi** to exit. If you have modified the file you are looking at currently and haven't written it out, **vi** will refuse to exit unless the "!" is supplied.

:e[!] [+[cmd]] [file]

Start editing a new file called "file" or start editing the current file over again. The command ":e!" says "ignore the changes I've made to this file and start over from the beginning". It is useful if you really mess up the file. The optional "+" says instead of starting at the beginning, start at the "end", or, if "cmd" is supplied, execute "cmd" first. Useful cases of this are where cmd is "n" (any integer) which starts at line number n, and "/text", which searches for "text" and starts at the line where it is found.

^^ Switch back to the place you were before your last tag command. If your last tag command stayed within the file, ^^ returns to that tag. If you have no recent tag command, it will return to the same place in the previous file that it was showing when you switched to the current file.

:n[!] Start editing the next file in the argument list. Since **vi** can be called with multiple file names, the ":n" command tells it to stop work on the current file and switch to the next file. If the current file was modifies, it has to be written out before the ":n" will work or else the "!" must be supplied, which says discard the changes I made to the current file.

:n[!] file [file file ...]

Replace the current argument list with a new list of files and start editing the first file in this new list.

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

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

:!cmd — Execute any UNIX shell command.

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

tag filename **vi**-search-command

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

6. Special Arrangements for Startup

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

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

* On version 6 systems: Instead of EXINIT, put the startup commands in the file .exrc in your home directory.

Should you inadvertently hang up the phone while inside **vi**, or should the computer crash, all may not be lost. Upon returning to the system, type:

```
vi –r file
```

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

7. Set Commands

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

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

and turned off with the command:

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

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

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

To display the value of a specific option type:

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

To display only those that you have changed type:

```
:set<nl>
```

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

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

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

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

environment, e.g.

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

or

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

for **sh** and **csh**, respectively. These are usually placed in your .profile or .login. If you are running a system without environments (such as version 6) you can place the set command in the file .exrc in your home directory.

autoindent
ai
Default: noai Type: toggle

When in autoindent mode, vi helps you indent code by starting each line in the same column as the preceding line. Tabbing to the right with <tab> or <^T> will move this boundary to the right, and it can be moved to the left with <^D>.

autoprint
ap
Default: ap Type: toggle

Causes the current line to be printed after each ex text modifying command. This is not of much interest in the normal **vi** visual mode.

autowrite
aw
Default: noaw Type: toggle

Autowrite causes an automatic write to be done if there are unsaved changes before certain commands which change files or otherwise interact with the outside world. These commands are :!, :tag, :next, :rewind, ^^, and ^].

beautify
bf
Default: nobf Type: toggle

Causes all control characters except <tab>, <nl>, and <ff> to be discarded.

directory
dir
Default: dir=/tmp Type: string

This is the directory in which **vi** puts its temporary file.

errorbells
eb
Default: noeb Type: toggle

Error messages are preceded by a <bell>.

hardtabs
ht

Default: hardtabs=8 Type: numeric

This option contains the value of hardware tabs in your terminal, or of software tabs expanded by the Unix system.

ignorecase
ic

Default: noic Type: toggle

All upper case characters are mapped to lower case in regular expression matching.

lisp

Default: nolisp Type: toggle

Autoindent for **lisp** code. The commands ( ) [[and]] are modified appropriately to affect s-expressions and functions.

list

Default: nolist Type: toggle

All printed lines have the <tab> and <nl> characters displayed visually.

magic

Default: magic Type: toggle

Enable the metacharacters for matching. These include **. * < > [string] [^string]** and **[<chr>-<chr>]**.

number
nu

Default: nonu Type: toggle

Each line is displayed with its line number.

open

Default: open Type: toggle

When set, prevents entering open or visual modes from ex or edit. Not of interest from vi.

optimize
opt

Default: opt Type: toggle

Basically of use only when using the **ex** capabilities. This option prevents automatic <cr>s from taking place, and speeds up output of indented lines, at the expense of losing typeahead on some versions of UNIX.

paragraphs
para

Default: para=IPLPPPQPP bp Type: string

Each pair of characters in the string indicate **nroff** macros which are to be treated as the beginning of a paragraph for the { and } commands. The default string is for the **-ms** and **-mm** macros. To indicate one letter **nroff** macros, such as **.P** or **.H**, quote a space in for the second character position. For example:

:set paragraphs=P\ bp<nl>

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

prompt — Default: prompt Type: toggle

In **ex** command mode the prompt character **:** will be printed when **ex** is waiting for a command. This is not of interest from vi.

redraw — Default: noredraw Type: toggle

On dumb terminals, force the screen to always be up to date, by sending great amounts of output. Useful only at high speeds.

report — Default: report=5 Type: numeric

This sets the threshold for the number of lines modified. When more than this number of lines are modified, removed, or yanked, **vi** will report the number of lines changed at the bottom of the screen.

scroll — Default: scroll={1/2 window} Type: numeric

This is the number of lines that the screen scrolls up or down when using the <^U> and <^D> commands.

sections — Default: sections=SHNHH HU Type: string

Each two character pair of this string specify **nroff** macro names which are to be treated as the beginning of a section by the]] and [[commands. The default string is for the **-ms** and **-mm** macros. To enter one letter **nroff** macros, use a quoted space as the second character. See **paragraphs** for a fuller explanation.

shell
sh — Default: sh=from environment SHELL or /bin/sh Type: string

This is the name of the **sh** to be used for "escaped" commands.

shiftwidth
sw — Default: sw=8 Type: numeric

This is the number of spaces that a <^T> or <^D> will move over for indenting, and the amount < and > shift by.

showmatch
sm

Default: nosm Type: toggle

When a) or } is typed, show the matching (or { by moving the cursor to it for one second if it is on the current screen.

slowopen
slow

Default: terminal dependent Type: toggle

On terminals that are slow and unintelligent, this option prevents the updating of the screen some of the time to improve speed.

tabstop
ts

Default: ts=8 Type: numeric

<tab>s are expanded to boundaries that are multiples of this value.

taglength
tl

Default: tl=0 Type: numeric

If nonzero, tag names are only significant to this many characters.

term

Default: (from environment TERM, else dumb) Type: string

This is the terminal and controls the visual displays. It cannot be changed when in "visual" mode, you have to Q to command mode, type a set term command, and do "vi." to get back into visual. Or exit vi, fix $TERM, and reenter. The definitions that drive a particular terminal type are found in the file **/etc/termcap**.

terse

Default: terse Type: toggle

When set, the error diagnostics are short.

warn

Default: warn Type: toggle

The user is warned if she/he tries to escape to the shell without writing out the current changes.

window

Default: window={8 at 600 baud or less, at 1200 baud, and screen size – 1 at 2400 baud or more} Type: numeric

This is the number of lines in the window whenever **vi** must redraw an entire screen. It is useful to make this size smaller if you are on a slow line.

w300
w1200
w9600

These set window, but only within the corresponding speed ranges. They are useful in an EXINIT to fine tune window sizes. For example,

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

causes a 4 lines window at speed up to 600 baud, a 12 line window at 1200 baud, and a full screen (the default) at over 1200 baud.

wrapscan
ws

Default: ws Type: toggle

Searches will wrap around the end of the file when is option is set. When it is off, the search will terminate when it reaches the end or the beginning of the file.

wrapmargin
wm

Default: wm=0 Type: numeric

Vi will automatically insert a <nl> when it finds a natural break point (usually a <sp> between words) that occurs within "wm" spaces of the right margin. Therefore with "wm=0" the option is off. Setting it to 10 would mean that any time you are within 10 spaces of the right margin **vi** would be looking for a <sp> or <tab> which it could replace with a <nl>. This is convenient for people who forget to look at the screen while they type. (In version 3, wrapmargin behaves more like nroff, in that the boundary specified by the distance from the right edge of the screen is taken as the rightmost edge of the area where a break is allowed, instead of the leftmost edge.)

writeany
wa

Default: nowa Type: toggle

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

6

SED — A Non-interactive Text Editor

Abstract

Sed is a non-interactive context editor that runs on the UNIX operating system. *Sed* is 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.

This memorandum constitutes a manual for users of *sed*.

Introduction

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

---

Source: Lee E. McMahon, *SED — A Non-interactive Text Editor* (Murray Hill, N.J.: Bell Laboratories, 1978).

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 any interactive editor known to the author, even if that editor can be driven by a pre-written script.

The principal loss of functions compared to an interactive editor are lack of relative addressing (because of the line-at-a-time operation), and lack of immediate verification that a command has done what was intended.

Sed is a lineal descendant of the UNIX editor, *ed*. Because of the differences between interactive and non-interactive operation, considerable changes have been made between *ed* and *sed*; even confirmed users of *ed* will frequently be surprised (and probably chagrined), if they rashly use *sed* without reading Sections 2 and 3 of this document. The most striking family resemblance between the two editors is in the class of patterns ("regular expressions") they recognize; the code for matching patterns is copied almost verbatim from the code for *ed*, and the description of regular expressions in Section 2 is copied almost verbatim from the UNIX Programmer's Manual[1]. (Both code and description were written by Dennis M. Ritchie.)

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; see Section 1.1 below.

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 in Section 2. Any number of blanks or tabs may separate the addresses from the function. The function must be present; the available commands are discussed in Section 3. The arguments may be required or optional, according to which function is given; again, they are discussed in Section 3 under each individual function.

Tab characters and spaces at the beginning of lines are ignored.

1.1 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 (see Section 3.3);
- **-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.

1.2 Order of Application of Editing Commands

Before any editing is done (in fact, before any input file is even opened), all the editing commands are compiled into a form which will be moderately 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* (see Section 3). 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.

1.3 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 (Section 3.6.).

1.4 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.
```

(In no case is the output of the *sed* commands to be considered an improvement on Coleridge.)

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. Addresses: Selecting Lines for Editing

Lines in the input file(s) to which editing commands are to be applied can be selected by addresses. Addresses 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 ('{ }')(Sec. 3.6.).

2.1 Line-number Addresses

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

As a special case, the character $ matches the last line of the last input file.

2.2 Context Addresses

A context address is a pattern ("regular expression") enclosed in slashes ('/'). The regular expressions recognized by *sed* are constructed as follows:

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 *except* 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 *s* command below and specification 10) immediately below.

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

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

NOTE

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.

2.3 Number of Addresses

The commands in the next section can have 0, 1, or 2 addresses. Under each command the maximum number of allowed addresses is given. For a command to have more addresses than the maximum allowed is considered an error.

If a command has no addresses, it is applied to every line in the input.

If a command has one address, it is applied to all lines 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.

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

3. 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.

3.1 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 the corpse.

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. (The *r* function is described in Section 3.4.)

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.

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
```

3.2 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 (see 2.2 above). 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 *d*th substring matched by parts of <pattern> enclosed in '\(' and '\)'. If nested substrings occur in <pattern>, the *d*th 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, 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.

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
```

3.3 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.)

Examples:

Assume that the file "note1" has the following contents:

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

Then the following command:

```
/Kubla/r note1
```

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.

3.4 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.

3.5 Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

(2)h hold pattern space

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

(2)H Hold pattern space

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

(2)g get contents of hold area

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

(2)G Get contents of hold area

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

(2)x exchange

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

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
```

3.6 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.

3.7 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.

Reference

[1] Ken Thompson and Dennis M. Ritchie, *The UNIX Programmer's Manual*. Bell Laboratories, 1978.

7

NROFF/TROFF User's Manual

Introduction

NROFF and TROFF are text processors under the UNIX Time-Sharing System [1] that format text for typewriter-like terminals and for a Graphic Systems phototypesetter, respectively. (Device-independent TROFF, part of the Documenter's Workbench, supports additional output devices.) 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.

Source: Joseph F. Ossanna (updated for 4.3BSD by Mark Seiden), *NROFF/TROFF User's Manual* (Murray Hill, N.J.: Bell Laboratories, 1976).

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*
–i	Read standard input after the input files are exhausted.
–m*name*	Prepends the macro file **/usr/lib/tmac.***name* to the input ***files***.
–n*N*	Number first generated page ***N***.
–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.
–q	Invoke the simultaneous input-output mode of the **rd** request.
–r*aN*	Number register ***a*** (one-character) is set to ***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.
–z	Efficiently suppress formatted output. Only produce output to standard error (from **tm** requests or diagnostics).

NROFF Only

–e Produce equally-spaced words in adjusted lines, using full terminal resolution.

–h On output, use tabs during horizontal spacing to increase speed. Device tabs setting are assumed to be (and input tabs are initially set to) every 8 character widths.

–T*name* Specifies the name of the output terminal type. Currently defined names are **37** for the (default) Model 37 Teletype®, **tn300** for the GE TermiNet 300 (or any terminal without half-line capabilities), **300S** for the DASI-300S, **300** for the DASI-300, and **450** for the DASI-450 (Diablo Hyterm).

TROFF Only

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

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

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

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

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

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. TC [4] is a 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* | **tc**

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 TC.

The remainder of this manual consists of: a Summary and Outline; a Reference Manual keyed to the outline; and a set of Tutorial Examples. Another tutorial is [5].

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.

[3] M. E. Lesk, *Tbl—A Program to Format Tables*, Bell Laboratories internal memorandum.

[4] Internal on-line documentation (*man* pages) on UNIX.

[5] B. W. Kernighan, *A TROFF Tutorial*, Bell Laboratories.

Summary of Requests

Font and Character Size Control

Request Form	Initial Value*	If No Argument	Explanation & Notes#
.ps ±*N*	10 point	previous	Point size; also \s±*N*.† [See Note E]
.fz *F* ±*N*	off	-	Font *F* to point size ±*N*. [See Note E]
.fz S *F* ±*N*	off	-	Special Font characters to point size ±*N*. [See Note E]
.ss *N*	12/36 em	ignored	Space-character size set to *N*/36 em.† [See Note E]
.cs *F N M*	off	-	Constant character space (width) mode (font *F*).† [See Note P]
.bd *F N*	off	-	Embolden font *F* by *N*−1 units.† [See Note P]
.bd S *F N*	off	-	Embolden Special Font when current font is *F*.† [See Note P]
.ft *F*	Roman	previous	Change to font *F* = *x*, *xx*, or 1-4. Also \f*x*, \f(*xx*, \f*N*. [See Note E]
.fp *N F*	R,I,B,S	ignored	Font named *F* mounted on physical position 1≤*N*≤4.

Page Control

Request Form	Initial Value*	If No Argument	Explanation & Notes#
.pl ±*N*	11 in	11 in	Page length. [See Note v]
.bp ±*N*	*N*=1	-	Eject current page; next page number *N*. [See Notes B‡,v]

* Values separated by ";" are for NROFF and TROFF respectively.
\# Notes are explained at the end of this Summary.
† No effect in NROFF.
‡ The use of " ´ " as control character (instead of ".") suppresses the break function.

.pn ±*N*	*N*=1	ignored	Next page number *N*.
.po ±*N*	0; 26/27 in	previous	Page offset. **[See Note v]**
.ne *N*	-	*N*=1*V*	Need *N* vertical space (*V* = vertical spacing). **[See Notes D,v]**
.mk *R*	none	internal	Mark current vertical place in register *R*. **[See Note D]**
.rt ±*N*	none	internal	Return *(upward only)* to marked vertical place. **[See Notes D,v]**

Text Filling, Adjusting and Centering

Request Form	Initial Value*	If No Argument	Explanation & Notes#
.br	-	-	Break. **[See Note B]**
.fi	ll	-	Fill output lines. **[See Notes B,E]**
.nf	fill	-	No filling or adjusting of output lines. **[See Notes B,E]**
.ad *c*	adj,both	adjust	Adjust output lines with mode *c*. **[See Note E]**
.na	adjust	-	No output line adjusting. **[See Note E]**
.ce *N*	off	*N*=1	Center following *N* input text lines. **[See Notes B,E]**

Vertical Spacing

Request Form	Initial Value*	If No Argument	Explanation & Notes#
.vs *N*	1/6in;12pts	previous	Vertical base line spacing (*V*). **[See Notes E,p]**

* Values separated by **";"** are for NROFF and TROFF respectively.
\# Notes are explained at the end of this Summary.

.ls *N*	*N*=1	previous	Output *N*−1 *V*s after each text output line. **[See Note E]**
.sp *N*	-	*N*=1*V*	Space vertical distance *N in either direction*. **[See Notes B,v]**
.sv *N*	-	*N*=1*V*	Save vertical distance *N*. **[See Note v]**
.os	-	-	Output saved vertical distance.
.ns	space	-	Turn no-space mode on. **[See Note D]**
.rs	-	-	Restore spacing; turn no-space mode off. **[See Note D]**

Line Length and Indenting

Request Form	Initial Value*	If No Argument	Explanation & Notes#
.ll ±*N*	6.5 in	previous	Line length. **[See Notes E,m]**
.in ±*N*	*N*=0	previous	Indent. **[See Notes B,E,m]**
.ti ±*N*	-	ignored	Temporary indent. **[See Notes B,E,m]**

Macros, Strings, Diversions and Position Traps

Request Form	Initial Value*	If No Argument	Explanation & Notes#
.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*.
.rm *xx*	-	ignored	Remove request, macro, or string.
.rn *xx yy*	-	ignored	Rename request, macro, or string *xx* to *yy*.
.di *xx*	-	end	Divert output to macro *xx*. **[See Note D]**
.da *xx*	-	end	Divert and append to *xx*. **[See Note D]**

* Values separated by ";" are for NROFF and TROFF respectively.
\# Notes are explained at the end of this Summary.

.wh *N xx*	-	-	Set location trap; negative is w.r.t. page bottom. **[See Note v]**
.ch *xx N*	-	-	Change trap location. **[See Note v]**
.dt *N xx*	-	off	Set a diversion trap. **[See Notes D,v]**
.it *N xx*	-	off	Set an input-line count trap. **[See Note E]**
.em *xx*	none	none	End macro is *xx*.

Number Registers

Request Form	Initial Value*	If No Argument	Explanation & Notes#
.nr *R ±N M*	-	-	Define and set number register *R*; auto-increment by *M*. **[See Note u]**
.af *R c*	arabic	-	Assign format to register *R* (*c*=**1**, **i**, **I**, **a**, **A**).
.rr *R*	-	-	Remove register *R*.

Tabs, Leaders and Fields

Request Form	Initial Value*	If No Argument	Explanation & Notes#
.ta *Nt ...*	0.8; 0.5in	none	Tab settings; *left* type, unless *t*=**R**(right), **C**(centered). **[See Notes E,m]**
.tc *c*	none	none	Tab repetition character. **[See Note E]**
.lc *c*	.	none	Leader repetition character. **[See Note E]**
.fc *a b*	off	off	Set field delimiter *a* and pad character *b*.

* Values separated by ";" are for NROFF and TROFF respectively.
\# Notes are explained at the end of this Summary.

Input and Output Conventions and Character Translations

Request Form	Initial Value*	If No Argument	Explanation & Notes#
.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	Underline (italicize in TROFF) *N* input lines. [See Note E]
.cu *N*	off	*N*=1	Continuous underline in NROFF; like **ul** in TROFF. [See Note E]
.uf *F*	Italic	Italic	Underline font set to *F* (to be switched to by **ul**).
.cc *c*	.	.	Set control character to *c*. [See Note E]
.c2 *c*	'	'	Set nobreak control character to *c*. [See Note E]
.tr *abcd*....	none	-	Translate *a* to *b*, etc. on output. [See Note O]

Hyphenation

Request Form	Initial Value*	If No Argument	Explanation & Notes#
.nh	hyphenate	-	No hyphenation. [See Note E]
.hy *N*	hyphenate	hyphenate	Hyphenate; *N* = mode. [See Note E]
.hc *c*	\%	\%	Hyphenation indicator character *c*. [See Note E]
.hw *word1* ...		ignored	Exception words.

* Values separated by ";" are for NROFF and TROFF respectively.
\# Notes are explained at the end of this Summary.

Three Part Titles

Request Form	Initial Value*	If No Argument	Explanation & Notes#
.tl *'left 'center 'right '*	-	-	Three part title.
.pc *c*	%	off	Page number character.
.lt ±*N*	6.5 in	previous	Length of title. [**See Notes E,m**]

Output Line Numbering

Request Form	Initial Value*	If No Argument	Explanation & Notes#
.nm ±*N M S I*	-	off	Number mode on or off, set parameters. [**See Note E**]
.nn *N*	-	*N*=1	Do not number next *N* lines. [**See Note E**]

Conditional Acceptance of Input

Request Form	Explanation & Notes#
.if *c anything*	If condition *c* true, accept *anything* as input, for multi-line use \{*anything* \}.
.if !*c anything*	If condition *c* false, accept *anything*.
.if *N anything*	If expression *N* > 0, accept *anything*. [**See Note u**]
.if !*N anything*	If expression *N* ≤ 0, accept *anything*. [**See Note u**]
.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*	If portion of if-else; all above forms (like **if**). [**See Note u**]
.el *anything*	Else portion of if-else.

* Values separated by ";" are for NROFF and TROFF respectively.
\# Notes are explained at the end of this Summary.

Environment Switching

Request Form	Initial Value*	If No Argument	Explanation & Notes#
.ev *N*	*N*=0	previous	Environment switched (*push down*).

Insertions from the Standard Input

Request Form	Initial Value*	If No Argument	Explanation & Notes#
.rd *prompt*	-	*prompt*=BEL	Read insertion.
.ex	-	-	Exit from NROFF/TROFF.

Input/Output File Switching

Request Form	Initial Value*	If No Argument	Explanation & Notes#
.so *filename*	-	-	Switch source file *(push down)*.
.nx *filename*	-	end-of-file	Next file.
.pi *program*	-	-	Pipe output to *program* (NROFF only).

Miscellaneous

Request Form	Initial Value*	If No Argument	Explanation & Notes#
.mc *c N*	-	off	Set margin character *c* and separation *N*. **[See Notes E,m]**
.tm *string*	-	newline	Print *string* on terminal (UNIX standard error output).
.ig *yy*	-	*.yy*=..	Ignore till call of *yy*.

* Values separated by ";" are for NROFF and TROFF respectively.
\# Notes are explained at the end of this Summary.

.pm *t*	-	all	Print macro names and sizes; if *t* present, print only total of sizes.
.ab *string*	-	-	Print a message and abort.
.fl	-	-	Flush output buffer. [**See Note B**]

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

ab	20	c2	10	di	7	ex	18	hw	13	lg	10	ne	3	os	5	rd	18	ss	2	uf	10
ad	4	cc	10	ds	7	fc	9	hy	13	li	10	nf	4	pc	14	rm	7	sv	5	ul	10
af	8	ce	4	dt	7	fi	4	ie	16	ll	6	nh	13	pi	19	rn	7	ta	9	vs	5
am	7	ch	7	ec	10	fl	20	if	16	ls	5	nm	15	pl	3	rr	8	tc	9	wh	7
as	7	cs	2	el	16	fp	2	ig	20	lt	14	nn	15	pm	20	rs	5	ti	6		
bd	2	cu	10	em	7	ft	2	in	6	mc	20	nr	8	pn	3	rt	3	tl	14		
bp	3	da	7	eo	10	fz	2	it	7	mk	3	ns	5	po	3	so	19	tm	20		
br	4	de	7	ev	17	hc	13	ie	9	na	4	nx	19	ps	2	sp	5	tr	10		

Escape Sequences for Characters, Indicators, and Functions

Section Reference	*Escape Sequence*	*Meaning*
10.1	\\	\ (to prevent or delay the interpretation of \ )
10.1	**\e**	Printable version of the *current* escape character.
2.1	\´	´ (acute accent); equivalent to **\(aa**
2.1	\`	` (grave accent); equivalent to **\(ga**
2.1	**\-**	- Minus sign in the *current* font
7	**\.**	Period (dot) (see **de**)
11.1	\(space)	Unpaddable space-size space character
11.1	**\0**	Digit width space
11.1	\ |	1/6 em narrow space character (zero width in NROFF)
11.1	\^	1/12 em half-narrow space character (zero width in NROFF)
4.1	**\&**	Non-printing, zero width character
10.6	**\!**	Transparent line indicator
10.7	**\"**	Beginning of comment
7.3	**\$***N*	Interpolate argument $1 \leq N \leq 9$
13	**\%**	Default optional hyphenation character
2.1	\(*xx*	Character named *xx*
7.1	**x*, *(*xx*	Interpolate string *x* or *xx*
9.1	**\a**	Non-interpreted leader character
12.3	**\b**´*abc...*´	Bracket building function
4.2	**\c**	Interrupt text processing
11.1	**\d**	Forward (down) 1/2 em vertical motion (1/2 line in NROFF)
2.2	**\f***x*,**\f(***xx*,**\f***N*	Change to font named *x* or *xx*, or position *N*
11.1	**\h**´*N*´	Local horizontal motion; move right *N (negative left)*
11.3	**\k***x*	Mark horizontal *input* place in register *x*
12.4	**\l**´*Nc*´	Horizontal line drawing function (optionally with *c*)
12.4	**\L**´*Nc*´	Vertical line drawing function (optionally with *c*)
8	**\n***x*,**\n(***xx*	Interpolate number register *x* or *xx*
12.1	**\o**´*abc...*´	Overstrike characters *a, b, c, ...*
4.1	**\p**	Break and spread output line
11.1	**\r**	Reverse 1 em vertical motion (reverse line in NROFF)
2.3	**\s***N*, **\s**±*N*	Point-size change function
9.1	**\t**	Non-interpreted horizontal tab
11.1	**\u**	Reverse (up) 1/2 em vertical motion (1/2 line in NROFF)
11.1	**\v**´*N*´	Local vertical motion; move down *N (negative up)*
11.2	**\w**´*string*´	Interpolate width of *string*
5.2	**\x**´*N*´	Extra line-space function *(negative before, positive after)*
12.2	**\z***c*	Print *c* with zero width (without spacing)
16	**\{**	Begin conditional input
16	**\}**	End conditional input
10.7	\(newline)	Concealed (ignored) newline
-	*X*	*X*, any character *not* listed above

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.
19	**c.**	Number of *lines* read from current input file.
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 (not in ditroff)
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.
5.3	**.L**	Set to current *line-spacing* (**ls**) parameter
-	**.P**	Set to 1 if the current page is being printed; otherwise 0.
-	**.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***'**.
19	**.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.
4.2	**.j**	Current adjustment mode and type.
4.1	**.k**	Length of text portion on current partial output line.
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 sthetic 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 (insertion in place) 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, (for historical reasons, corresponding to the Graphic Systems phototypesetter which had 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 its own internal horizontal/vertical resolution. 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 scale indicator suffixes 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 prefixed 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 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. $\pm N$ means that the argument may take the forms N, $+N$, or $-N$ and that the corresponding effect is to set the affected parameter to N, to increment it by N, or to decrement it by N respectively. Plain N means that an initial algebraic sign is *not* an increment indicator, but merely the sign of N. Generally, unreasonable numerical input is either ignored or truncated to a reasonable value. For example, most requests expect to set parameters to non-negative values; exceptions are **sp**, **wh**, **ch**, **nr**, and **if**. The requests **ps**, **ft**, **po**, **vs**, **ls**, **ll**, **in**, and **lt** restore the *previous* parameter value in the *absence* of an argument.

Single character arguments are indicated by single lower case letters and one/two character arguments are indicated by a pair of lower case letters. Character string arguments are indicated by multi-character mnemonics.

2. Font and Character Size Control

2.1 Character Set

The TROFF character set consists of a typesetter-dependent basic character set plus a Special Mathematical Font character set—each having 102 characters. An example of these character sets is shown in the Appendix Table I. All printable ASCII characters are included, with some on the Special Font. With three exceptions, these 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 Appendix 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 are typesetter dependent, but often include 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* ($1 \leq N \leq 9$) to increment/decrement the size by *N*; **\s0** restores the *previous* size. Requested point size values that are between two valid sizes yield the larger of the two. The current size is available in the **.s** register. NROFF ignores type size control.

Request Form	Initial Value	If No Argument	Explanation & Notes*
.ps ±*N*	10 point	previous	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. **[See Note E]**

* Notes are explained at the end of the Summary above.

.fz *F* ±*N*	off	-	The characters in font *F* will be adjusted to be in size ±*N*. Characters in the Special Font encountered during the use of font *F* will have the same size modification. (Use the **.fz S** request if different treatment of Special Font characters is required). **.fz** must follow any **.fp** request for the position. **[See Note E]**
.fz S *F* ±*N*	off	-	The characters in the Special Font will be in size ±*N* independent of previous **.fz** requests. **[See Note E]**
.ss *N*	12/36 em	ignored	Space-character size is set to *N*/36 ems. This size is the minimum word spacing in adjusted text. Ignored in NROFF. **[See Note E]**
.cs *FNM*	off	-	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. **[See Note P]**
.bd *F N*	off	-	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 mode must be still or again in effect when the characters are physically printed. Ignored in NROFF. **[See Note P]**
.bd S *F N*	off	-	The characters in the Special Font will be emboldened whenever the current font is *F*. The mode must be still or again in effect when the characters are physically printed. **[See Note P]**

.ft *F*	Roman	previous	Font changed to *F*. Alternatively, imbed \f*F*. The font name **P** is reserved to mean the previous font. [See Note E]
.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 usable page width on the Graphic Systems phototypesetter was about 7.54 inches, beginning about 1/27 inch from the left edge of the 8 inch wide, continuous roll paper, but these characteristics are typesetter-dependent. The physical limitations on NROFF output are output-device dependent.

Request Form	Initial Value	If No Argument	Explanation & Notes*
.pl ±*N*	11 in	11 in	Page length set to ±*N*. The internal limitation is about 75 inches in TROFF and about 136 inches in NROFF. The current page length is available in the **.p** register. [**See Note v**]
.bp ±*N*	*N*=1	-	Begin page. The current page is ejected and a new page is begun. If ±*N* is given, the new page number will be ±*N*. Also see request **ns**. [**See Notes B#,v**]
.pn ±*N*	*N*=1	ignored	Page number. The next page (when it occurs) will have the page number ±*N*. A **pn** must occur before the initial pseudo-page transition to affect the page number of the first page. The current page number is in the % register.
.po ±*N*	0; 26/27 in†	previous	Page offset. The current *left margin* is set to ±*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. [**See Note v**]
.ne *N*	-	*N*=1 *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 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. [**See Notes D,v**]
.mk *R*	none	internal	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. [**See Note D**]

* Notes are explained at the end of the Summary above.
\# The use of " ´ " as control character (instead of ".") suppresses the break function.
† Values separated by ";" are for NROFF and TROFF respectively.

.rt ±*N*	none	internal	Return *upward only* to a marked vertical place in the current diversion. If ±*N* (w.r.t. current place) is given, the place is ±*N* from the top of the page or diversion or, if *N* is absent, to a place marked by a previous **mk**. Note that the **sp** request (§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**. **[See Notes D,v]**

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 to hyphenate the word 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. The **.k** register (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.

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 (§10.4) can be made to not look like a control line by preceding it by 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	Explanation & Notes*
.br	-	-	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. [**See Note B**]
.fi	fill on	-	Fill subsequent output lines. The register **.u** is 1 in fill mode and 0 in nofill mode. [**See Notes B,E**]
.nf	fill on	-	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. [**See Notes B,E**]

* Notes are explained at the end of the Summary above.

.ad *c*	adj,both	adjust	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. The type indicator can also be a value saved from the read-only **.j** number register, which is set to contain the current adjustment mode and type. **[See Note E]**

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	-	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. **[See Note E]**
.ce *N*	off	*N*=1	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. **[See Notes B,E]**

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	Explanation & Notes*
.vs *N*	1/6in; 12pts†	previous	Set vertical base-line spacing size *V*. Transient *extra* vertical space available with **\x**´*N*´ (see above). **[See Notes E,p]**
.ls *N*	*N*=1	previous	*Line* spacing set to ±*N*. *N*−1 *V*s *(blank lines)* are appended to each output text line. The (read-only) number register **.L** is set to contain the current line-spacing value. Appended blank lines are omitted, if the text or previous appended blank line reached a trap position. **[See Note E]**

* Notes are explained at the end of the Summary above.
† Values separated by ";" are for NROFF and TROFF respectively.

.sp *N*	-	*N*=1*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). **[See Notes B,v]**
.sv *N*	-	*N*=1*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*. **[See Note v]**
.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	-	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**. **[See Note D]**
.rs	space	-	Restore spacing. The no-space mode is turned off. **[See Note D]**
Blank text line.			Causes a break and outputs a blank line just like **sp 1**. **[See Note B]**

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	Explanation & Notes*
.ll ±*N*	6.5 in	previous	Line length is set to ±*N*. In TROFF the maximum (line-length)+(page-offset) is about 7.54 inches. **[See Notes E,m]**
.in ±*N*	*N*=0	previous	Indent is set to ±*N*. The indent is prepended to each output line. **[See Notes B,E,m]**
.ti ±*N*	-	ignored	Temporary indent. The *next* output text line will be indented a distance ±*N* with respect to the current indent. The resulting total indent may not be negative. The current indent is not changed. **[See Notes B,E,m]**

7. Macros, Strings, Diversions 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.

* Notes are explained at the end of the Summary above.

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 ($1 \leq N \leq 9$). If an invoked argument doesn't exist, a null string results. For example, the macro *xx* may be defined by

```
.de xx          \"begin definition
Today is \\$1 the \\$2.
..              \"end definition
```

and called by

```
.xx Monday 14th
```

to produce the text

Today is Monday the 14th.

Note that the \$ was concealed in the definition with a prepended \. The number of currently available arguments is in the **.$** register.

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 the **it** request below.

Request Form	Initial Value	If No Argument	Explanation & Notes*
.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	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. **[See Note D]**
.da *xx*	-	end	Divert, appending to *xx* (append version of **di**). **[See Note D]**

* Notes are explained at the end of the Summary above.

.wh *N xx*	-	-	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. [**See Note v**]
.ch *xx N*	-	-	Change the trap position for macro *xx* to be *N*. In the absence of *N*, the trap, if any, is removed. [**See Note v**]
.dt *N xx*	-	off	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. [**See Notes D,v**]
.it *N xx*	-	off	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. [**See Note E**]
.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). 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	Explanation & Notes*
.nr *R* ±*N M*	-	-	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*. [See Note u]
.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.

* Notes are explained at the end of the Summary above.

.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	Explanation & Notes*
.ta *Nt* ...	8n; 0.5in#	none	Set tab stops and types. *t*=**R**, right adjusting; *t*=**C**, centering; *t* absent, left adjusting. TROFF tab stops are preset every 0.5in.; NROFF every 8 character widths. The stop values are separated by spaces, and a value preceded by + is treated as an increment to the previous stop value. **[See Notes E,m]**
.tc *c*	none	none	The tab repetition character becomes *c*, or is removed specifying motion. **[See Note E]**
.lc *c*	.	none	The leader repetition character becomes *c*, or is removed specifying motion. **[See Note E]**
.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.

* Notes are explained at the end of the Summary above.
\# Values separated by ";" are for NROFF and TROFF respectively.

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. \ 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	Explanation & Notes*
.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.

* Notes are explained at the end of the Summary above.

Request Form	Initial Value	If No Argument	Explanation & Notes*
.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 Times Italic on font position 2 (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	Explanation & Notes*
.cu *N*	off	*N*=1	A variant of **ul** that causes *every* character to be underlined in NROFF. Identical to **ul** in TROFF. **[See Note E]**
.uf *F*	Italic	Italic	Underline font set to *F*. In NROFF, *F* may *not* be on position 1 (initially Times Roman).

* Notes are explained at the end of the Summary above.

.ul *N*	off	*N*=1	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. **[See Note E]**

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	Explanation & Notes*
.cc *c*	.	.	The basic control character is set to *c*, or reset to ".". **[See Note E]**
.c2 *c*	'	'	The *nobreak* control character is set to *c*, or reset to "'". **[See Note E]**

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).

* Notes are explained at the end of the Summary above.

Request Form	Initial Value	If No Argument	Explanation & Notes*
.tr *abcd....*	none	-	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. **[See Note O]**

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

* Notes are explained at the end of the Summary above.

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 tables.

Vertical Local Motion	Effect in TROFF	NROFF
\v'*N*'	Move distance *N*	
\u	½ em up	½ line up
\d	½ em down	½ line down
\r	1 em up	1 line up

Horizontal Local Motion	Effect in TROFF	NROFF
\h'*N*'	Move distance *N*	
\(space)	Unpaddable space-size space	
\0	Digit-size space	
\\|	1/6 em space	ignored
\^	1/12 em space	ignored

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 **\kx***word***\h'** | **\nxu+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* are 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 root-en ‾, 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 ul
\\$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
\"next line draws box
\h'-.5n'\L' \\nau-1'\l'\\n(.lu+1n\(ul'\L'- \\nau+1'\l' 0u-.5n\(ul'
.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	Explanation & Notes*
.nh	hyphen-ate	-	Automatic hyphenation is turned off. [**See Note E**]

* Notes are explained at the end of the Summary above.

.hy*N*	on, *N*=1	on,*N*=1	Automatic hyphenation is turned on for $N \geq 1$, 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. **[See Note E]**
.hc *c*	\%	\%	Hyphenation indicator character is set to *c* or to the default \%. The indicator does not appear in the output. **[See Note E]**
.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	Explanation & Notes*
.pc *c*	%	off	The page number character is set to *c*, or removed. The page-number register remains %.
.lt ±*N*	6.5 in	previous	Length of title set to ±*N*. The line-length and the title-length are *independent*. Indents do not apply to titles; page-offsets do. **[See Notes E,m]**

* Notes are explained at the end of the Summary above.

.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.

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 3 digit-space is prepended to output text lines. The text lines are thus offset by four digit-spaces, and otherwise retain their line length; a reduction in line length may be desired to keep the 6 right margin aligned with an earlier margin. Blank lines, other vertical spaces, and lines generated by **tl** are *not* numbered. Numbering can be temporarily suspended with **nn**, or with an 9 **.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 12 numbers that are multiples of some number *M* are to be printed (the others will appear as blank number fields).

Request Form	Initial Value	If No Argument	Explanation & Notes*
.nn *N*	-	*N*=1	The next *N* text output lines are not numbered. [See Note E]

* Notes are explained at the end of the Summary above.

.nm ±*N M S I*	off	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**. [See Note E]

As an example, the paragraph portions of this section are numbered with *M*= 3: **.nm 1 3** was placed at the beginning; **.nm** was placed at the end of the first paragraph; and **.nm +0** was placed in front of this paragraph; and **.nm** finally placed at the end. Line lengths were also changed (by **\w'0000'u**) to keep the right side aligned. Another example is **.nm +5 5 x 3** which turns on numbering with the line number of the next line to be 5 greater than the last numbered line, with *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	Explanation & Notes*
.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*	If expression *N* > 0, accept *anything*. [See Note u]
.if !*N anything*	If expression *N* ≤ 0, accept *anything*. [See Note u]

* Notes are explained at the end of the Summary above.

.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*	If portion of if-else; all above forms (like **if**). **[See Note u]**
.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	Explanation & Notes*
.ev *N*	*N*=0	previous	Environment switched to environment 0≤*N*≤2. Switching is done in push-down fashion so that restoring a previous environment *must* be done with **.ev** rather than specific reference.

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	Explanation & Notes*
.ex	-	-	Exit from NROFF/TROFF. Text processing is terminated exactly as if all input had ended.

* Notes are explained at the end of the Summary above.

.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*.

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

The (read-only) number register **.c** contains the input line number in the current input file. The number register **c.** is a general register serving the same purpose.

Request Form	Initial Value	If No Argument	Explanation & Notes*
.so *filename*		-	Switch source file. The top input (file reading) level is switched to *filename*. The effect of an **so** encountered in a macro occurs immediately. 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*.

* Notes are explained at the end of the Summary above.

.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	Explanation & Notes*
.mc *c N*	-	off	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 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. [**See Notes E,m**]
.tm *string*	-	newline	After skipping initial blanks, *string* (rest of the line) is read in *copy mode* and written on the user's terminal. (see §21).
.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.

* Notes are explained at the end of the Summary above.

.ab *string*	-	-	Print *string* on standard error and terminate immediately. The default *string* is "User Abort". Does not cause a break. Only output preceding the last break is written.
.fl	-	-	Flush output buffer. Used in interactive debugging to force output.

21. Output and Error Messages

The output from **tm**, **pm**, **ab** and the prompt from **rd**, as well as various *error* messages are written onto UNIX's *standard error* 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

---

* 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).

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 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 n 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 ¼em space (all measurements on 8.5 × 11 inch paper prior to photoreduction). This font sample is printed on an Imagen Printer, using the Image Network Computer Modern fonts. These fonts are similar to the Times fonts which were used in the original implementations of troff.

Computer Modern Roman

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |
● □ — - _ ¼ ½ ¾ ﬁ ﬂ ﬀ ﬃ ﬄ ° † ′ ¢ ® ©

Computer Modern Italic

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
*! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |*
● □ — - _ ¼ ½ ¾ *ﬁ ﬂ ﬀ ﬃ ﬄ* ° † ′ ¢ ® ©

Computer Modern 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

Char	*Input Name*	*Character Name*
fi	\(fi	fi
fl	\(fl	fl
ff	\(ff	ff
ffi	\(Fi	ffi
ffl	\(Fl	ffl
°	\(de	degree
†	\(dg	dagger
′	\(fm	foot mark
¢	\(ct	cent sign
®	\(rg	registered
©	\(co	copyright

Non-ASCII characters and ´, `, _, +, −, =, and * on the special font.

The ASCII characters &, #, ", ´, `, <, >, \, {, }, ~, ^, and _ exist *only* on the special font and are printed as a 1-em space if that font is not mounted. The following characters exist only on the special font except for the upper case Greek letter names followed by † which are mapped into upper case English letters in whatever font is mounted on font position one (default Times Roman). The special math plus, minus, and equals are provided to insulate the appearance of equations from the choice of standard fonts.

Char	*Input Name*	*Character Name*
+	\(pl	math plus
−	\(mi	math minus
=	\(eq	math equals
*	\(**	math star
§	\(sc	section
´	\(aa	acute accent
`	\(ga	grave accent
_	\(ul	underrule
/	\(sl	slash (matching backslash)

Char	*Input Name*	*Character Name*
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

Char	Input Name	Character Name
κ	\(*k	kappa
λ	\(*l	lambda
μ	\(*m	mu
ν	\(*n	nu
ξ	\(*c	xi
o	\(*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
A	\(*A	Alpha†
B	\(*B	Beta†
Γ	\(*G	Gamma
Δ	\(*D	Delta
E	\(*E	Epsilon†
Z	\(*Z	Zeta†
H	\(*Y	Eta†
Θ	\(*H	Theta
I	\(*I	Iota†
K	\(*K	Kappa†
Λ	\(*L	Lambda
M	\(*M	Mu†
N	\(*N	Nu†
Ξ	\(*C	Xi
O	\(*O	Omicron†
Π	\(*P	Pi
P	\(*R	Rho†
Σ	\(*S	Sigma
T	\(*T	Tau†
Y	\(*U	Upsilon
Φ	\(*F	Phi
X	\(*X	Chi†
Ψ	\(*Q	Psi
Ω	\(*W	Omega
√	\(sr	square root
⎷	\(rn	root en extender
≥	\(>=	>=
≤	\(<=	<=
≡	\(==	identically equal
≃	\(~=	approx =
~	\(ap	approximates
≠	\(!=	not equal
→	\(->	right arrow
←	\(<-	left arrow
↑	\(ua	up arrow
↓	\(da	down arrow
×	\(mu	multiply
÷	\(di	divide
±	\(+-	plus-minus
∪	\(cu	cup (union)
∩	\(ca	cap (intersection)
⊂	\(sb	subset of
⊃	\(sp	superset of
⊆	\(ib	improper subset
⊇	\(ip	improper superset
∞	\(if	infinity
∂	\(pd	partial derivative
∇	\(gr	gradient
¬	\(no	not
∫	\(is	integral sign
∝	\(pt	proportional to
∅	\(es	empty set
∈	\(mo	member of
|	\(br	box vertical rule
‡	\(dd	double dagger
☞	\(rh	right hand
☜	\(lh	left hand
Ⓑ	\(bs	Bell System logo (typesetter-dependent)
|	\(or	or
○	\(ci	circle
⎧	\(lt	left top of big curly bracket
⎩	\(lb	left bottom
⎫	\(rt	right top
⎭	\(rb	right bottom
⎨	\(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)

8

A TROFF Tutorial

Abstract

troff is a text-formatting program for typesetting on the UNIX operating system. 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 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, well as the usual features of a formatter—as 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** and a more recent version *(device-independent* **troff**) are identical to the version described here, so this document also serves as a tutorial for them as well.

Source: Brian W. Kernighan (updated for 4.3BSD by Mark Seiden), *A TROFF Tutorial* (Murray Hill, N.J.: Bell Laboratories, 1978).

1. Introduction

troff [1] is a text-formatting program, written originally by J. F. Ossanna, for producing high-quality printed output from the phototypesetter on the UNIX operating system. 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], PWB/MM [5], and "–me" [6] packages for internal memoranda and external papers provide most of the facilities needed for a wide range of document preparation. There are also packages for viewgraphs, for simulating the older **roff** formatters, 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 supplied with UNIX Version 7 and 32V 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 16
Some more text.
```

will change the "point size", that is, the size of the letters being printed, to "16 point" (one point is 1/72 inch) like this:

Some text. Some more text.

Occasionally, though, something special occurs in the middle of a line—to produce

Area = πr^2

you have to type

```
Area = \(*p\fIr\fR\ \s8\u2\d\s0
```

(which we will explain shortly). The backslash character \ 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 jugs.
11 point: Pack my box with five dozen liquor jugs.
12 point: Pack my box with five dozen liquor jugs.
14 point: Pack my box with five dozen liquor jugs.
16 point 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. The original of this document (on 8.5 by 11 inch paper) is in 12 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

* This font is not 22 points since there is no 22-point font available in the software used to produce this document.

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 "12 on 14," that is,

```
.ps 12
.vs 14p
```

If we changed to

```
.ps 12
.vs 12p
```

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 14 on 16.

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. (The complete list of font sizes and styles depends on your typesetter or laser printer.) 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	$\times$
\(*b	β
)	)
\(->	$\rightarrow$
\(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, which some people think is 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, this line was produced 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:

Pater noster qui est in caelis sanctificetur nomen tuum; adveniat regnum tuum; fiat voluntas tua, sicut in caelo, et in terra. ... Amen.

Of course, there is also some 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). Also, the line **'in -.3i** is inserted just before "fiat" to move the indent so that "caelo" lines up under the "P".

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 ****. (**\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 –2.25i         (shorten lines)
.ti –0.3i          (move P back)
\36\v'2'P\v'-2\s0ater 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

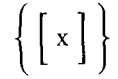
; instead of ; or ;

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 experimentally-derived 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:



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 tlphone, 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 so easily in **troff**, a serious hardship for the novice. Instead you have to do a lot of specification (or use a macro package, which makes it effortless).

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. 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 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, 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 o .tl 'odd page title'- % -'
.if e .tl '- % -'even page title'
```

gives facing pages different titles and page numbers on the outside edge 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.

[6] Eric P. Allman, *Writing Papers with NROFF Using -me*, University of California, Berkeley.

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[1]	-	\(hy		

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	s̲	\(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:

"	{	}	<	>	~	^	\	#	◘

For greek, precede the roman letter by \(* to get the corresponding greek; for example, \(***a** is α.[2]

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
A	B	Γ	Δ	E	Z	H	Θ	I	K	Λ	M	N	Ξ	O	Π	P	Σ	T	Y	Φ	X	Ψ	Ω	

1. In some character sets, the bold \(sq produces a solid (filled) box.
2. If a greek character does not exist on the special font, the equivalent character in the current font is printed.

9

MM — Memorandum Macros

1. Introduction

1.1 Purpose

This memorandum is the user's guide and reference manual for the Memorandum Macros (MM), a general-purpose package of text formatting macros for use with the UNIX text formatters *nroff* and *troff*.

The purpose of MM is to provide a unified, consistent, and flexible tool for producing many common types of documents. Although the UNIX time-sharing system provides other macro packages for various *specialized* formats, MM has become the standard, general-purpose macro package for most documents.

MM can be used to produce:

- Letters
- Reports
- Technical Memoranda
- Released Papers
- Manuals
- Books.

The uses of MM range from single-page letters to documents of several hundred pages in length, such as user guides, design proposals, etc.

Source: D.W. Smith, J.R. Mashey, E.C. Pariser (January 1980 Revision), and N.W. Smith (June 1980 Revision), *MM — Memorandum Macros* Piscataway, N.J.: Bell Laboratories).

1.2 Conventions

Each section of this memorandum explains a single facility of MM. In general, the earlier a section occurs, the more necessary it is for most users. Some of the later sections can be completely ignored if MM defaults are acceptable. Likewise, each section progresses from normal-case to special-case facilities. We recommend reading a section in detail only until there is enough information to obtain the desired format, then skimming the rest of it, because some details may be of use to just a few people.

Numbers enclosed in curly brackets ({}) refer to section numbers within this document. For example, this is {1.2}.

Sections that require knowledge of the formatters {1.4} have a bullet (•) at the end of their headings.

In the synopses of macro calls, square brackets ([]) surrounding an argument indicate that it is optional. Ellipses (...) show that the preceding argument may appear more than once.

A reference of the form *name*(N) points to page *name* in section *N* of the *UNIX User's Manual* [1].

The examples of *output* in this manual are as produced by *troff; nroff* output would, of course, look somewhat different (Appendix C shows both *nroff* and *troff* outputs for a simple letter). In those cases in which the behavior of the two formatters is truly different, the *nroff* action is described first, with the *troff* action following in parentheses. For example:

The title is underlined (italic).

means that the title is underlined in *nroff* and italic in *troff*.

1.3 Overall Structure of a Document

The input for a document that is to be formatted with MM possesses four major segments, any of which may be omitted; if present, they *must* occur in the following order:

Parameter-setting — This segment sets the general style and appearance of a document. The user can control page width, margin justification, numbering styles for headings and lists, page headers and footers {9}, and many other properties of the document. Also, the user can add macros or redefine existing

ones. This segment can be omitted entirely if one is satisfied with default values; it produces no actual output, but only performs the setup for the rest of the document.

Beginning — This segment includes those items that occur only once, at the beginning of a document, e.g., title, author's name, date.

Body — This segment is the actual text of the document. It may be as small as a single paragraph, or as large as hundreds of pages. It may have a hierarchy of *headings* up to seven levels deep {4}. Headings are automatically numbered (if desired) and can be saved to generate the table of contents. Five additional levels of subordination are provided by a set of *list* macros for automatic numbering, alphabetic sequencing, and "marking" of list items {5}. The body may also contain various types of displays, tables, figures, references, and footnotes {7, 8, 11}.

Ending — This segment contains those items that occur once only, at the end of a document. Included here are signature(s) and lists of notations (e.g., "copy to" lists) {6.11}. Certain macros may be invoked here to print information that is wholly or partially derived from the rest of the document, such as the table of contents or the cover sheet for a document {10}.

The existence and size of these four segments varies widely among different document types. Although a specific item (such as date, title, author name(s), etc.) may be printed in several different ways depending on the document type, there is a uniform way of typing it in.

1.4 Definitions

The term *formatter* refers to either of the text-formatting programs *nroff* and *troff*.

Requests are built-in commands recognized by the formatters. Although one seldom needs to use these requests directly {3.10}, this document contains references to some of them. Full details are given in the *NROFF/TROFF User's Manual*[2]. For example, the request:

```
.sp
```

inserts a blank line in the output.

Macros are named collections of requests. Each macro is an abbreviation for a collection of requests that would otherwise require repetition. MM supplies many macros, and the user can define additional ones. Macros and requests share the same set of names and are used in the same way.

Strings provide character variables, each of which names a string of characters. Strings are often used in page headers, page footers, and lists. They share the pool of names used by *requests* and *macros*. A string can be given a value via the .ds (define string) request, and its value can be obtained by referencing its name, preceded by "*" (for 1-character names) or "*(" (for 2-character names). For instance, the string *DT* in MM normally contains the current date, so that the *input* line:

```
Today is \*(DT.
```

may result in the following *output:*

```
Today is May 1, 1917
```

The current date can be replaced, e.g.:

```
.ds DT 01/01/79
```

or by invoking a macro designed for that purpose {6.7.1}.

Number registers fill the role of integer variables. They are used for flags, for arithmetic, and for automatic numbering. A register can be given a value using a .nr request, and be referenced by preceding its name by "\n" (for 1-character names) or "\n(" (for 2-character names). For example, the following sets the value of the register *d* to 1 more than that of the register *dd*:

```
.nr d 1+\n(dd
```

See {14.1} regarding naming conventions for requests, macros, strings, and number registers. Appendix E list all macros, strings, and number registers defined in MM.

1.5 Prerequisites and Further Reading

1.5.1 Prerequisites

We assume familiarity with UNIX at the level given in *UNIX for Beginners*[3] and *A Tutorial Introduction to the UNIX Text Editor*[4]. Some familiarity with the request summary in the *NROFF/TROFF User's Manual*[2] is helpful.

1.5.2 Further Reading

NROFF/TROFF User's Manual[2] provides detailed descriptions of formatter capabilities, while *A TROFF Tutorial*[5] provides a general overview. See *Typesetting Mathematics—User's Guide*[6] for instructions on formatting mathematical expressions. See *tbl*(1) and *TBL—A Program to Format Tables*[7] for instructions on formatting tabular data.

Examples of formatted documents and of their respective input, as well as a quick reference to the material in this manual are given in *Typing Documents with MM*[8].

2. Invoking the Macros

This section tells how to access MM, shows UNIX command lines appropriate for various output devices, and describes command-line flags for MM.

2.1 The mm Command

The *mm*(1) command can be used to print documents using *nroff* and MM; this command invokes *nroff* with the –cm flag {2.2}. It has options to specify preprocessing by *tbl* and/or by *neqn* and for postprocessing by various output filters. Any arguments or flags that are not recognized by *mm*(1), e.g. –rC3, are passed to *nroff* or to MM, as appropriate. The options, which can occur in any order but *must* appear before the file names, are:

–e	*neqn* is to be invoked; also causes *neqn* to read */usr/pub/eqnchar* (see *eqnchar*(7)).
–t	*tbl*(1) is to be invoked.
–c	*col*(1) is to be invoked.
–E	the –e option of *nroff* is to be invoked.
–y	–mm (uncompacted macros) is to be used instead of –cm.
–12	12-pitch mode is to be used. Be sure the pitch switch on the terminal is set to 12.
–T450	output is to a DASI 450. This is the *default* terminal type (unless $TERM is set). It is also equivalent to –T1620.
–T450–12	output is to a DASI 450 in 12-pitch mode.
–T300	output is to a DASI 300 terminal.
–T300–12	output is to a DASI 300 in 12-pitch mode.
–T300s	output is to a DASI 300S.
–T300s–12	output is to a DASI 300S in 12-pitch mode.
–T4014	output is to a Tektronix 4014.
–T37	output is to a *TELETYPE*® Model 37.
–T382	output is to a DTC-382.
–T4000a	output is to a Trendata 4000A.
–TX	output is prepared for an EBCDIC line printer.
–Thp	output is to a HP264x (implies –c).
–T43	output is to a *TELETYPE* Model 43 (implies –c).
–T40/4	output is to a *TELETYPE* Model 40/4 (implies –c).
–T745	output is to a Texas Instrument 700 series terminal (implies –c).
–T2631	output is prepared for a HP2631 printer (where –T2631–e and –T2631–c may be used for expanded and compressed modes, respectively) (implies –c).
–Tlp	output is to a device with no reverse or partial line motions or other special features (implies –c).

Any other –T option given does not produce an error; it is equivalent to –Tlp.

A similar command is available for use with *troff* (see *mmt*(1)).

2.2 The –cm or –mm Flag

The MM package can also be invoked by including the –cm or –mm flag as an argument to the formatter. The –cm flag causes the pre-compacted version of the macros to be loaded. The –mm flag causes the file /usr/lib/tmac/tmac.m to be read and processed before any other files. This action defines the MM macros, sets default values for various parameters, and initializes the formatter to be ready to process the files of input text.

2.3 Typical Command Lines

The prototype command lines are as follows (with the various options explained in {2.4} and in the *NROFF/TROFF User's Manual*[2]).

- Text without tables or equations:
  mm [options] file-name ...
  or **nroff** [options] **–cm** file-name ...
  mmt [options] file-name ...
  or **troff** [options] **–cm** file-name ...
- Text with tables:
  mm –t [options] file-name ...
  or **tbl** file-name ... | **nroff** [options] **–cm**
  mmt –t [options] file-name ...
  or **tbl** file-name ... | **troff** [options] **–cm**
- Text with equations:
  mm –e [options] file-name ...
  or **neqn /usr/pub/eqnchar** file-name ... | **nroff** [options] **–cm**
  mmt –e [options] file-name ...
  or **eqn /usr/pub/eqnchar** file-name ... | **troff** [options] **–cm**
- Text with both tables and equations:
  mm –t –e [options] file-name ...
  or **tbl** file-name ... | **neqn /usr/pub/eqnchar** – | **nroff** [options] **–cm**
  mmt –t –e [options] file-name ...
  or **tbl** file-name ... | **eqn /usr/pub/eqnchar** – | **troff** [options] **–cm**

When formatting a document with *nroff*, the output should normally be processed for a specific type of terminal, because the output may

require some features that are specific to a given terminal, e.g., reverse paper motion or half-line paper motion in both directions. Some commonly-used terminal types and the command lines appropriate for them are given below. See {2.4} as well as *300*(1), *450*(1), *4014*(1), *hp*(1), *col*(1), and *terminals*(7) for further information.

- DASI 450 in 10-pitch, 6 lines/inch mode, with 0.75 inch offset, and a line length of 6 inches (60 characters) where this is the default terminal type so no –T option is needed (unless $TERM is set to another value):
  mm file-name ...
  or **nroff –T450 –h –cm** file-name ...
- DASI 450 in 12-pitch, 6 lines/inch mode, with 0.75 inch offset, and a line length of 6 inches (72 characters):
  mm –12 file-name ...
  or **nroff –T450–12 –h –cm** file-name ...
  or, to increase the line length to 80 characters and decrease the offset to 3 characters:
  mm –12 –rW80 –rO3 file-name ...
  or **nroff –T450–12 –rW80 –rO3 –h –cm** file-name ...
- Hewlett-Packard HP264x CRT family:
  mm –Thp file-name ...
  or **nroff –cm** file-name ... **| col | hp**
- Any terminal incapable of reverse paper motion and also lacking hardware tab stops (Texas Instruments 700 series, etc.):
  mm –T745 file-name ...
  or **nroff –cm** file-name ... **| col –x**
- Versatec printer (see *vp*(1) for additional details):
  vp [vp-options] "**mm –rT2 –c** file-name ..."
  or **vp** [vp-options] "**nroff –rT2 –cm** file-name ... **| col**"

Of course, *tbl*(1) and *eqn*(1)/*neqn*, if needed, must be invoked as shown in the command line prototypes at the beginning of this section.

If two-column processing {12.4} is used with *nroff*, either the –c option must be specified to *mm*(1),[1] or the *nroff* output must be

1. Note that *mm*(1) uses *col*(1) automatically for many of the terminal types {2.1}.

postprocessed by *col*(1). In the latter case, the –T37 terminal type must be specified to *nroff*, the –h option must *not* be specified, and the output of *col*(1) must be processed by the appropriate terminal filter (e.g., *450*(1)); *mm*(1) with the –c option handles all this automatically.

2.4 Parameters that Can Be Set from the Command Line

Number registers are commonly used within MM to hold parameter values that control various aspects of output style. Many of these can be changed within the text files via .nr requests. In addition, some of these registers can be set from the command line itself, a useful feature for those parameters that should *not* be permanently embedded within the input text itself. If used, these registers (with the possible exception of the register *P*— see below) *must* be set on the command line (or before the MM macro definitions are processed) and their meanings are:

–rA*n* for *n* = 1 has the effect of invoking the .AF macro without an argument {6.7.2}. If *n* = 2 allows for usage of the Bell System logo, if available, on a printing device (currently available for the Xerox 9700 only).

–rC*n* *n* sets the type of copy (e.g., DRAFT) to be printed at the bottom of each page. See {9.5}.
n = 1 for OFFICIAL FILE COPY.
n = 2 for DATE FILE COPY.
n = 3 for DRAFT with single-spacing and default paragraph style.
n = 4 for DRAFT with double-spacing and 10 space paragraph indent.

–rD1 sets *debug mode*. This flag requests the formatter to attempt to continue processing even if MM detects errors that would otherwise cause termination. It also includes some debugging information in the default page header {9.2, 12.3}.

–rE*n* controls the font of the Subject/Date/From fields. If *n* is 1 then these fields are bold (default for *troff*) and if *n* is 0 then these fields are roman (regular text–default for *nroff*).

–rL*k* sets the length of the physical page to *k* lines.[2] The default

value is 66 lines per page. This parameter is used, for example, when directing output to a Versatec printer.

–r*N*n specifies the page numbering style. When *n* is 0 (default), all pages get the (prevailing) header {9.2}. When *n* is 1, the page header replaces the footer on page 1 only. When *n* is 2, the page header is omitted from page 1. When *n* is 3, "section-page" numbering {4.5} occurs (see .FD {8.3} and .RP {11.4} for footnote and reference numbering in sections). When *n* is 4, the *default* page header is suppressed; however a user-specified header is not affected. When *n* is 5, "section-page" and "section-figure" numbering occurs.

n	*Page 1*	*Pages 2ff.*
0	header	header
1	header replaces footer	header
2	no header	header
3	"section-page" as footer	same as page 1
4	no header	no header unless .PH defined
5	same as 3-with "section-figure"	same as page 1

The contents of the prevailing header and footer do *not* depend on the value of the number register *N*; *N* only controls whether and where the header (and, for *N*=3 or 5, the footer) is printed, as well as the page numbering style. In particular, if the header and footer are null {9.2, 9.5}, the value of *N* is irrelevant.

–r*O*k offsets output *k* spaces to the right.[2] It is helpful for adjusting output positioning on some terminals. The default offset if this register is not set on the command line is 0.75 inches.

NOTE

The register name is the capital letter "O", *not* the digit zero (0).

2. For *nroff*, *k* is an *unscaled* number representing lines or character positions; for *troff*, *k* must be *scaled*.

–rP*n* specifies that the pages of the document are to be numbered starting with *n*. This register may also be set via a .nr request in the input text.

–rS*n* sets the point size and vertical spacing for the document. The default *n* is 10, i.e., 10-point type on 12-point vertical spacing, giving 6 lines per inch {12.9}. This parameter applies to *troff* only.

–rT*n* provides register settings for certain devices. If *n* is 1, then the line length and page offset are set to 80 and 3, respectively. Setting *n* to 2 changes the page length to 84 lines per page and inhibits underlining; it is meant for output sent to the Versatec printer. The default value for *n* is 0. This parameter applies to *nroff* only.

–rU1 controls underlining of section headings. This flag causes only letters and digits to be underlined. Otherwise, all characters (including spaces) are underlined {4.2.2.4.2}. This parameter applies to *nroff* only.

–rW*k* page width (i.e., line length and title length) is set to *k*.[2] This can be used to change the page width from the default value of 6 inches (60 characters in 10 pitch or 72 characters in 12 pitch).

2.5 Omission of –cm or –mm

If a large number of arguments is required on the command line, it may be convenient to set up the first (or only) input file of a document as follows:

```
zero or more initializations of registers listed in {2.4}
.so /usr/lib/tmac/tmac.m
remainder of text
```

In this case, one must *not* use the –cm or –mm flag (nor the *mm*(1) or *mmt*(1) command); the .so request has the equivalent effect, but the registers in {2.4} must be initialized *before* the .so request, because their values are meaningful only if set before the macro definitions are processed. When using this method, it is best to "lock" into the input file only those parameters that are seldom changed. For example:

```
.nr W 80
.nr O 10
.nr N 3
.so /usr/lib/tmac/tmac.m
.H 1 "INTRODUCTION"
   :
   :
```

specifies, for *nroff*, a line length of 80, a page offset of 10, and "section-page" numbering.

3. Formatting Concepts

3.1 Basic Terms

The normal action of the formatters is to *fill* output lines from one or more input lines. The output lines may be *justified* so that both the left and right margins are aligned. As the lines are being filled, words may also be hyphenated {3.4} as necessary. It is possible to turn any of these modes on and off (see .SA {12.2}, *Hy* {3.4}, and the formatter .nf and .fi requests[2]). Turning off fill mode also turns off justification and hyphenation.

Certain formatting commands (requests and macros) cause the filling of the current output line to cease, the line (of whatever length) to be printed, and the subsequent text to begin a new output line. This printing of a partially filled output line is known as a *break*. A few formatter requests and most of the MM macros cause a break.

While formatter requests can be used with MM, one must fully understand the consequences and side-effects that each such request might have. Actually, there is little need to use formatter requests; the macros described here should be used in most cases because:

— it is much easier to control (and change at any later point in time) the overall style of the document.

— complicated features (such as footnotes or tables of contents) can be obtained with ease.

— the user is insulated from the peculiarities of the formatter language.

A good rule is to use formatter requests only when absolutely necessary {3.10}.

In order to make it easy to revise the input text at a later time, input lines should be kept short and should be broken at the end of clauses; each new full *sentence must* begin on a new line.

3.2 Arguments and Double Quotes

For any macro call, a *null argument* is an argument whose width is zero. Such an argument often has a special meaning; the preferred form for a null argument is `" "`. Note that *omitting* an argument is *not* the same as supplying a *null argument* (for example, see the .MT macro in ⟨6.6⟩). Furthermore, omitted arguments can occur only at the end of an argument list, while null arguments can occur anywhere.

Any macro argument containing ordinary (paddable) spaces *must* be enclosed in double quotes (").[3] Otherwise, it will be treated as several separate arguments.

Double quotes (") are *not* permitted as part of the value of a macro argument or of a string that is to be used as a macro argument. If you must, use two grave accents (``) and/or two acute accents ('') instead. This restriction is necessary because many macro arguments are processed (interpreted) a variable number of times; for example, headings are first printed in the text and may be (re)printed in the table of contents.

3.3 Unpaddable Spaces

When output lines are *justified* to give an even right margin, existing spaces in a line may have additional spaces appended to them. This may harm the desired alignment of text. To avoid this problem, it is necessary to be able to specify a space that cannot be expanded during justification, i.e., an *unpaddable space*. There are several ways to accomplish this.

3. A double quote (") is a *single* character that must not be confused with two apostrophes or acute accents (''), or with two grave accents (``).

First, one may type a backslash followed by a space ("\ "). This pair of characters directly generates an *unpaddable space*. Second, one may sacrifice some seldom-used character to be translated into a space upon output. Because this translation occurs after justification, the chosen character may be used anywhere an unpaddable space is desired. The tilde (~) is often used for this purpose. To use it in this way, insert the following at the beginning of the document:

```
.tr ~
```

If a tilde must actually appear in the output, it can be temporarily "recovered" by inserting:

```
.tr ~~
```

before the place where it is needed. Its previous usage is restored by repeating the ".tr ~", but only after a break or after the line containing the tilde has been forced out. Note that the use of the tilde in this fashion is *not* recommended for documents in which the tilde is used within equations.

3.4 Hyphenation

The formatters do not perform hyphenation unless the user requests it. Hyphenation can be turned on in the body of the text by specifying:

```
.nr Hy 1
```

once at the beginning of the document. For hyphenation within footnotes and across pages, see {8.3}.

If hyphenation is requested, the formatters will automatically hyphenate words, if need be. However, the user may specify the hyphenation points for a specific occurrence of any word by the use of a special character known as a hyphenation indicator, or may specify hyphenation points for a small list of words (about 128 characters).

If the *hyphenation indicator* (initially, the two-character sequence "\%") appears at the beginning of a word, the word is *not* hyphenated. Alternatively, it can be used to indicate legal hyphenation point(s) inside a word. In any case, *all* occurrences of the hyphenation indicator disappear on output.

The user may specify a different hyphenation indicator:

```
.HC [hyphenation-indicator]
```

The circumflex (^) is often used for this purpose; this is done by inserting the following at the beginning of a document:

```
.HC ^
```

Note that any word containing hyphens or dashes—also known as *em* dashes—will be hyphenated immediately after a hyphen or dash if it is necessary to hyphenate the word, *even if the formatter hyphenation function is turned off.*

The user may supply, via the .hw request, a small list of words with the proper hyphenation points indicated. For example, to indicate the proper hyphenation of the word "printout," one may specify:

```
.hw print-out
```

3.5 Tabs

The macros .MT {6.6}, .TC {10.1}, and .CS {10.2} use the formatter .ta request to set tab stops, and then restore the *default* values[4] of tab settings. Thus, setting tabs to other than the default values is the user's responsibility.

Note that a tab character is always interpreted with respect to its position on the *input line*, rather than its position on the output line. In general, tab characters should appear only on lines processed in "no-fill" mode {3.1}.

Also note that *tbl*(1) {7.3} changes tab stops, but does *not* restore the default tab settings.

3.6 Special Use of the BEL Character

The non-printing character BEL is used as a delimiter in many macros where it is necessary to compute the width of an argument or to delimit arbitrary text, e.g., in headers and footers {9}, headings {4}, and list marks {5}. Users who include BEL characters in their

4. Every eight characters in *nroff*; every ½ inch in *troff*.

input text (especially in arguments to macros) will receive mangled output.

3.7 Bullets

A bullet (•) is often obtained on a typewriter terminal by using an "o" overstruck by a "+". For compatibility with *troff*, a bullet string is provided by MM. Rather than overstriking, use the sequence:

```
\*(BU
```

wherever a bullet is desired. Note that the bullet list (.BL) macros {5.3.3.2} use this string to automatically generate the bullets for the list items.

3.8 Dashes, Minus Signs, and Hyphens

Troff has distinct graphics for a dash, a minus sign, and a hyphen, while *nroff* does not. Those who intend to use *nroff* only may use the minus sign ("–") for all three.

Those who wish mainly to use *troff* should follow the escape conventions of the *NROFF/TROFF User's Manual*[2].

Those who want to use both formatters must take care during text preparation. Unfortunately, these characters cannot be represented in a way that is both compatible and convenient. We suggest the following approach:

Dash — Type *(EM for each text dash for both *nroff* and *troff*. This string generates an em dash (—) in *troff* and " -- " in *nroff*. Note that the dash list (.DL) macros {5.3.3.3} automatically generate the em dash for each list item.

Hyphen — Type "–" and use as is for both formatters. *Nroff* will print it as is, and *troff* will print "-" (a true hyphen).

Minus — Type "\–" for a true minus sign, regardless of formatter. *Nroff* will effectively ignore the "\", while *troff* will print a true minus sign.

3.9 Trademark String

A trademark string *(Tm is available with MM. This places the letters "TM" one-half line above the text that it follows.

For example:

The UNIX*(Tm User's Manual is available from the library.

yields:

The UNIX™ User's Manual is available from the library.

3.10 Use of Formatter Requests

Most formatter requests[2] should *not* be used with MM because MM provides the corresponding formatting functions in a much more user-oriented and surprise-free fashion than do the basic formatter requests {3.1}. However, some formatter requests *are* useful with MM, namely:

.af .br .ce .de .ds .fi .hw .ls .nf .nr .nx
.rm .rr .rs .so .sp .ta .ti .tl .tr .!

The .fp, .lg, and .ss requests are also sometimes useful for *troff*. Use of other requests without fully understanding their implications very often leads to disaster.

4. Paragraphs and Headings

This section describes simple paragraphs and section headings. Additional paragraph and list styles are covered in {5}.

4.1 Paragraphs

.P [type]
one or more lines of text.

This macro is used to begin two kinds of paragraphs. In a *left-justified* paragraph, the first line begins at the left margin, while in an *indented* paragraph, it is indented five spaces (see below).

A document possesses a *default paragraph style* obtained by specifying ".P" before each paragraph that does *not* follow a heading {4.2}. The default style is controlled by the register *Pt*. The initial value of *Pt* is 0, which always provides left-justified paragraphs. All paragraphs can be forced to be indented by inserting the following at the beginning of the document:

```
.nr Pt 1
```

All paragraphs will be indented except after headings, lists, and displays if the following:

```
.nr Pt 2
```

is inserted at the beginning of the document.

The amount a paragraph is indented is contained in the register *Pi*, whose default value is 5. To indent paragraphs by, say, 10 spaces, insert:

```
.nr Pi 10
```

at the beginning of the document. Of course, both the *Pi* and *Pt* register values must be greater than zero for any paragraphs to be indented.

The number register *Ps* controls the amount of spacing between paragraphs. By default, *Ps* is set to 1, yielding one blank space (½ a vertical space).

CAUTION

Values that specify indentation must be *unscaled* and are treated as "character positions," i.e., as a number of *ens*. In *troff*, an *en* is the number of points (1 point = 1/72 of an inch) equal to half the current point size. In *nroff*, an *en* is equal to the width of a character.

Regardless of the value of *Pt*, an *individual* paragraph can be forced to be left-justified or indented. ".P 0" always forces left justification; ".P 1" always causes indentation by the amount specified by the register *Pi*.

If .P occurs inside a *list*, the indent (if any) of the paragraph is added to the current list indent {5}.

Numbered paragraphs may be produced by setting the register *Np* to 1. This produces paragraphs numbered within first level headings, e.g., 1.01, 1.02, 1.03, 2.01, etc.

A different style of numbered paragraphs is obtained by using the

```
.nP
```

macro rather than the .P macro for paragraphs. This produces paragraphs that are numbered within second level headings and contain a "double-line indent" in which the text of the second line is indented to be aligned with the text of the first line so that the number stands out.

```
.H 1 "FIRST HEADING"
.H 2 "Second Heading"
.nP
one or more lines of text
```

4.2 Numbered Headings

```
.H level [heading-text] [heading-suffix]
zero or more lines of text
```

The .H macro provides seven levels of numbered headings, as illustrated by this document. Level 1 is the most major or highest; level 7 the lowest.

The *heading-suffix* is appended to the *heading-text* and may be used for footnote marks which should not appear with the heading text in the Table of Contents.

CAUTION

Strictly speaking, there is no need for a .P macro immediately after a .H (or .HU {4.3}), because the .H macro also performs the function of the .P macro, and an immediately following .P is ignored {4.2.2.2}. It is, however, good practice to start *every* paragraph with a .P macro, thereby ensuring that all paragraphs uniformly begin with a .P throughout an entire document.

4.2.1 Normal Appearance

The normal appearance of headings is as shown in this document. The effect of .H varies according to the *level* argument. First-level headings are *preceded* by two blank lines (one vertical space); all others are *preceded* by one blank line (½ a vertical space).

.H 1 heading-text — gives a bold heading *followed* by a single blank line (½ a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Full capital letters should normally be used to make the heading stand out.

.H 2 heading-text — yields a bold heading followed by a single blank line (½ a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Normally, initial capitals are used.

.H *n* heading-text — for $3 \leq n \leq 7$, produces an underlined (italic) heading followed by two spaces. The following text appears on the same line, i.e., these are *run-in* headings.

Appropriate numbering and spacing (horizontal and vertical) occur even if the heading text is omitted from a .H macro call.

Here are the first few .H calls of {4}:

```
.H 1 "PARAGRAPHS AND HEADINGS"
.H 2 "Paragraphs"
.H 2 "Numbered Headings"
.H 3 "Normal Appearance."
.H 3 "Altering Appearance of Headings."
.H 4 "Pre-Spacing and Page Ejection."
.H 4 "Spacing After Headings."
.H 4 "Centered Headings."
.H 4 "Bold, Italic, and Underlined Headings"
.H 5 "Control by Level."
```

4.2.2 Altering Appearance of Headings

Users satisfied with the default appearance of headings may skip to {4.3}. One can modify the appearance of headings quite easily by setting certain registers and strings at the beginning of the document. This permits quick alteration of a document's style, because this style-control information is concentrated in a few lines, rather than being distributed throughout the document.

4.2.2.1 Pre-Spacing and Page Ejection

A first-level heading normally has two blank lines (one vertical space) preceding it, and all others have one blank line (½ a vertical space). If a multi-line heading were to be split across pages, it is automatically moved to the top of the next page. Every first-level heading may be forced to the top of a new page by inserting:

```
.nr Ej 1
```

at the beginning of the document. Long documents may be made more manageable if each section starts on a new page. Setting *Ej* to a higher value causes the same effect for headings up to that level, i.e., a page eject occurs if the heading level is less than or equal to *Ej*.

4.2.2.2 Spacing After Headings

Three registers control the appearance of text immediately following a .H call. They are *Hb* (heading break level), *Hs* (heading space level), and *Hi* (post-heading indent).

If the heading level is less than or equal to *Hb*, a break {3.1} occurs after the heading. If the heading level is less than or equal to *Hs*, a blank line (½ a vertical space) is inserted after the heading. Defaults for *Hb* and *Hs* are 2. If a heading level is greater than *Hb* and also greater than *Hs*, then the heading (if any) is run into the following text. These registers permit headings to be separated from the text in a consistent way throughout a document, while allowing easy alteration of white space and heading emphasis.

For any *stand-alone* heading, i.e., a heading not run into the following text, the alignment of the next line of output is controlled by the register *Hi*. If *Hi* is 0, text is left-justified. If *Hi* is 1 (the *default* value), the text is indented according to the paragraph type as specified by the register *Pt* {4.1}. Finally, if *Hi* is 2, text is indented to line up with the first word of the heading itself, so that the heading number stands out more clearly.

For example, to cause a blank line (½ a vertical space) to appear after the first three heading levels, to have no run-in headings, and to force the text following all headings to be left-justified (regardless of the value of *Pt*), the following should appear at the top of the document:

```
.nr Hs 3
.nr Hb7
.nr Hi 0
```

4.2.2.3 Centered Headings

The register *Hc* can be used to obtain centered headings. A heading is centered if its level is less than or equal to *Hc*, *and* if it is also stand-alone {4.2.2.2}. *Hc* is 0 initially (no centered headings).

4.2.2.4 Bold, Italic, and Underlined Headings

4.2.2.4.1 Control by Level

Any heading that is underlined by *nroff* is made italic by *troff*. The string *HF* (heading font) contains seven codes that specify the fonts for heading levels 1-7. The legal codes, their interpretations, and the defaults for *HF* are:

Formatter	*HF Code*			*Default*
	1	*2*	*3*	*HF*
nroff	no underline	underline	bold	3 3 2 2 2 2 2
troff	roman	italic	bold	3 3 2 2 2 2 2

Thus, levels 1 and 2 are bold; levels 3 through 7 are underlined in *nroff* and italic in *troff*. The user may reset HF as desired. Any value omitted from the right end of the list is taken to be 1. For example, the following would result in five bold levels and two non-underlined (roman) levels:

```
.ds HF 3 3 3 3 3
```

4.2.2.4.2 Nroff Underlining Style

Nroff can underline in two ways. The normal style (.ul request) is to underline only letters and digits. The continuous style (.cu request) underlines all characters, including spaces. By default, MM attempts to use the continuous style on any heading that is to be underlined and is short enough to fit on a single line. If a heading is to be underlined, but is too long, it is underlined the normal way (i.e., only letters and digits are underlined).

All underlining of headings can be forced to the normal way by using the –rU1 flag when invoking *nroff* {2.4}.

4.2.2.4.3 Heading Point Sizes

The user may also specify the desired point size for each heading level with the *HP* string (for use with *troff* only).

```
.ds HP [ps1] [ps2] [ps3] [ps4] [ps5] [ps6] [ps7]
```

By default, the text of headings (.H and .HU) is printed in the same point size as the body *except* that bold stand-alone headings are printed in a size one point smaller than the body. The string *HP*, similar to the string *HF*, can be specified to contain up to seven values, corresponding to the seven levels of headings. For example:

```
.ds HP 12 12 10 10 10 10 10
```

specifies that the first and second level headings are to be printed in 12-point type, with the remainder printed in 10-point. Note that the specified values may also be *relative* point-size changes, e.g.:

```
.ds HP +2 +2 -1 -1
```

If absolute point sizes are specified, then those sizes will be used regardless of the point size of the body of the document. If relative point sizes are specified, then the point sizes for the headings will be relative to the point size of the body, even if the latter is changed.

Null or zero values imply that the *default* size will be used for the corresponding heading level.

CAUTION

Only the point size of the headings is affected. Specifying a large point size without providing increased vertical spacing (via .HX and/or .HZ) may cause overprinting.

4.2.2.5 Marking Styles—Numerals and Concatenation

```
.HM [arg1] . . . [arg7]
```

The registers named *H1* through *H7* are used as counters for the seven levels of headings. Their values are normally printed using Arabic numerals. The .HM macro (heading mark style) allows this

choice to be overridden, thus providing "outline" and other document styles. This macro can have up to seven arguments; each argument is a string indicating the type of marking to be used. Legal values and their meanings are shown below; omitted values are interpreted as 1, while illegal values have no effect.

Value	*Interpretation*
1	Arabic (default for all levels)
0001	Arabic with enough leading zeroes to get the specified number of digits
A	Upper-case alphabetic
a	Lower-case alphabetic
I	Upper-case Roman
i	Lower-case Roman

By default, the complete heading mark for a given level is built by concatenating the mark for that level to the right of all marks for all levels of higher value. To inhibit the concatenation of heading level marks, i.e., to obtain just the current level mark followed by a period, set the register *Ht* (heading-mark type) to 1.

For example, a commonly-used "outline" style is obtained by:

```
.HM I A 1 a i
.nr Ht 1
```

4.3 Unnumbered Headings

.HU heading-text

.HU is a special case of .H; it is handled in the same way as .H, except that no heading mark is printed. In order to preserve the hierarchical structure of headings when .H and .HU calls are intermixed, each .HU heading is considered to exist at the level given by register *Hu*, whose initial value is 2. Thus, in the normal case, the only difference between:

.HU heading-text

and:

.H 2 heading-text

is the printing of the heading mark for the latter. Both have the

effect of incrementing the numbering counter for level 2, and resetting to zero the counters for levels 3 through 7. Typically, the value of *Hu* should be set to make unnumbered headings (if any) be the lowest-level headings in a document.

.HU can be especially helpful in setting up appendices and other sections that may not fit well into the numbering scheme of the main body of a document {14.2.1}.

4.4 Headings and the Table of Contents

The text of headings and their corresponding page numbers can be automatically collected for a table of contents. This is accomplished by doing the following two things:

- specifying in the register *Cl* what level headings are to be saved
- invoking the .TC macro {10.1} at the end of the document

Any heading whose level is less than or equal to the value of the register *Cl* (contents level) is saved and later displayed in the table of contents. The default value for *Cl* is 2, i.e., the first two levels of headings are saved.

Due to the way the headings are saved, it is possible to exceed the formatter's storage capacity, particularly when saving many levels of many headings, while also processing displays {7} and footnotes {8}. If this happens, the "Out of temp file space" diagnostic {Appendix D} will be issued; the only remedy is to save fewer levels and/or to have fewer words in the heading text.

4.5 First-Level Headings and Page Numbering Style

By default, pages are numbered sequentially at the top of the page. For large documents, it may be desirable to use page numbering of the form "section-page," where *section* is the number of the current first-level heading. This page numbering style can be achieved by specifying the –rN3 or –rN5 flag on the command line {9.9}. As a side effect, this also has the effect of setting *Ej* to 1, i.e., each section begins on a new page. In this style, the page number is printed at the *bottom* of the page, so that the correct section number is printed.

4.6 User Exit Macros

NOTE

This section is intended only for users who are accustomed to writing formatter macros.

.HX dlevel rlevel heading-text
.HY dlevel rlevel heading-text
.HZ dlevel rlevel heading-text

The .HX, .HY, and .HZ macros are the means by which the user obtains a final level of control over the previously-described heading mechanism. MM does not define .HX, .HY, and .HZ; they are intended to be defined by the user. The .H macro invokes .HX shortly before the actual heading text is printed; it calls .HZ as its last action. After .HX is invoked, the size of the heading is calculated. This processing causes certain features that may have been included in .HX, such as .ti for temporary indent, to be lost. After the size calculation, .HY is invoked so that the user may respecify these features. All the default actions occur if these macros are not defined. If the .HX, .HY, or .HZ are defined by the user, the user-supplied definition is interpreted at the appropriate point. These macros can therefore influence the handling of all headings, because the .HU macro is actually a special case of the .H macro.

If the user originally invoked the .H macro, then the derived level (*dlevel*) and the real level (*rlevel*) are both equal to the level given in the .H invocation. If the user originally invoked the .HU macro {4.3}, *dlevel* is equal to the contents of register *Hu*, and *rlevel* is 0. In both cases, *heading-text* is the text of the original invocation.

By the time .H calls .HX, it has already incremented the heading counter of the specified level {4.2.2.5}, produced blank line(s) (vertical space) to precede the heading {4.2.2.1}, and accumulated the "heading mark", i.e., the string of digits, letters, and periods needed for a numbered heading. When .HX is called, all user-accessible registers and strings can be referenced, as well as the following:

string }0 If *rlevel* is non-zero, this string contains the "heading mark." Two unpaddable spaces (to separate the *mark* from the *heading*) have been appended to this string. If *rlevel* is 0, this string is null.

register ;0 This register indicates the type of spacing that is to follow the heading {4.2.2.2}. A value of 0 means that the heading is run-in. A value of 1 means a break (but no blank line) is to follow the heading. A value of 2 means that a blank line (½ a vertical space) is to follow the heading.

string }2 If register ;0 is 0, this string contains two unpaddable spaces that will be used to separate the (run-in) *heading* from the following *text*. If register ;0 is non-zero, this string is null.

register ;3 This register contains an adjustment factor for a .ne request issued before the heading is actually printed. On entry to .HX, it has the value 3 if *dlevel* equals 1, and 1 otherwise. The .ne request is for the following number of lines: the contents of the register ;0 taken as blank lines (halves of vertical space) plus the contents of register ;3 as blank lines (halves of vertical space) plus the number of lines of the heading.

The user may alter the values of }0, }2, and ;3 within .HX as desired. The following are examples of actions that might be performed by defining .HX to include the lines shown:

Change first-level heading mark from format *n.* to *n*.0:
```
.if \\$1=1 .ds }0 \\n(H1.0\□\□          (□ stands for a space)
```

Separate run-in heading from the text with a period and two unpaddable spaces:
```
.if \\n(;0=0 .ds }2 .\□\□
```

Assure that at least 15 lines are left on the page before printing a first-level heading:
```
.if \\$1=1 .nr ;3 15-\\n(;0
```

Add 3 additional blank lines before each first-level heading:
```
.if \\$1=1 .sp 3
```

Indent level 3 run-in headings by 5 spaces:
```
.if \\$1=3 .ti 5n
```

If temporary strings or macros are used within .HX, chose their names with care {14.1}.

.HY is called after the .ne is issued. Certain features requested in .HX must be repeated. For example:

```
.de HY
.if \\$1=3 .ti 5n
..
```

.HZ is called at the end of .H to permit user-controlled actions after the heading is produced. For example, in a large document, sections may correspond to chapters of a book, and the user may want to change a page header or footer, e.g.:

```
.de HZ
.if \\$1=1 .PF " "Section \\$3" "
..
```

4.7 Hints for Large Documents

A large document is often organized for convenience into one file per section. If the files are numbered, it is wise to use enough digits in the names of these files for the maximum number of sections, i.e., use suffix numbers 01 through 20 rather than 1 through 9 and 10 through 20.

Users often want to format individual sections of long documents. To do this with the correct section numbers, it is necessary to set register *H1* to 1 less than the number of the section just *before* the corresponding ".H 1" call. For example, at the beginning of section 5, insert:

```
.nr H1 4
```

CAUTION

This is a dangerous practice: it defeats the automatic (re)numbering of sections when sections are added or deleted. Remove such lines as soon as possible.

5. Lists

This section describes many different kinds of lists: automatically-numbered and alphabetized lists, bullet lists, dash lists, lists with arbitrary marks, and lists starting with arbitrary strings, e.g., with terms or phrases to be defined.

5.1 Basic Approach

In order to avoid repetitive typing of arguments to describe the appearance of items in a list, MM provides a convenient way to specify lists. All lists are composed of the following parts:

- A *list-initialization* macro that controls the appearance of the list: line spacing, indentation, marking with special symbols, and numbering or alphabetizing.
- One or more *List Item* (.LI) macros, each followed by the actual text of the corresponding list item.
- The *List End* (.LE) macro that terminates the list and restores the previous indentation.

Lists may be nested up to six levels. The list-initialization macro saves the previous list status (indentation, marking style, etc.); the .LE macro restores it.

With this approach, the format of a list is specified only once at the beginning of that list. In addition, by building on the existing structure, users may create their own customized sets of list macros with relatively little effort {5.4, Appendix A}.

5.2 Sample Nested Lists

The input for several lists and the corresponding output are shown below. The .AL and .DL macro calls {5.3.3} contained therein are examples of the *list-initialization* macros. This example will help us to explain the material in the following sections. Input text:

```
.AL A
.LI
This is an alphabetized item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.AL
.LI
This is a numbered item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.DL
.LI
This is a dash item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LI + 1
This is a dash item with a "plus" as prefix.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LE
.LI
This is numbered item 2.
.LE
.LI
This is another alphabetized item, B.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LE
.P
This paragraph appears at the left margin.
```

Output:

A. This is an alphabetized item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

    1. This is a numbered item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

        — This is a dash item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

      + — This is a dash item with a "plus" as prefix. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

    2. This is numbered item 2.

B. This is another alphabetized item, B. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

This paragraph appears at the left margin.

5.3 Basic List Macros

Because all lists share the same overall structure except for the list-initialization macro, we first discuss the macros common to all lists. Each list-initialization macro is covered in {5.3.3}.

5.3.1 List Item

.LI [mark] [1]
one or more lines of text that make up the list item.

The .LI macro is used with all lists. It normally causes the output of a single blank line (½ a vertical space) before its item, although this may be suppressed. If no arguments are given, it labels its item with the *current mark*, which is specified by the most recent list-initialization macro. If a single argument is given to .LI, that argument is output *instead of* the current mark. If two arguments are given, the first argument becomes a *prefix* to the current mark, thus

allowing the user to emphasize one or more items in a list. One unpaddable space is inserted between the prefix and the mark. For example:

```
.BL 6
.LI
This is a simple bullet item.
.LI +
This replaces the bullet with a ''plus.''
.LI + 1
But this uses ''plus'' as prefix to the bullet.
.LE
```

yields:

- This is a simple bullet item.

+ This replaces the bullet with a "plus."

+ • But this uses "plus" as prefix to the bullet.

CAUTION

The *mark* must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified {3.3}.

If the *current mark* (in the *current list*) is a null string, and the first argument of .LI is omitted or null, the resulting effect is that of a *hanging indent*, i.e., the first line of the following text is "outdented," starting at the same place where the *mark* would have started {5.3.3.6}.

5.3.2 List End

.LE [1]

List End restores the state of the list back to that existing just before the most recent list-initialization macro call. If the optional argument is given, the .LE outputs a blank line (½ a vertical space). This option should generally be used only when the .LE is followed by running text, but not when followed by a macro that produces blank lines of its own, such as .P, .H, or .LI.

.H and .HU automatically clear all list information, so one may legally omit the .LE(s) that would normally occur just before either of these macros. Such a practice is *not* recommended, however, because errors will occur if the list text is separated from the heading at some later time (e.g., by insertion of text).

5.3.3 List Initialization Macros

The following are the various list-initialization macros. They are actually implemented as calls to the more basic .LB macro {5.4}.

5.3.3.1 Automatically-Numbered or Alphabetized Lists

.AL [type] [text-indent] [1]

The .AL macro is used to begin sequentially-numbered or alphabetized lists. If there are no arguments, the list is numbered, and text is indented *Li*, initially 6 (5)[5] spaces from the indent in force when the .AL is called, thus leaving room for a space, two digits, a period, and two spaces before the text.

Spacing at the beginning of the list and between the items can be suppressed by setting the *Ls* (list space) register. *Ls* is set to the innermost list level for which spacing *is* done. For example:

.nr Ls 0

specifies that no spacing will occur around *any* list items. The default value for *Ls* is 6 (which is the *maximum* list nesting level).

The *type* argument may be given to obtain a different type of sequencing, and its value should indicate the first element in the sequence desired, i.e., it must be 1, A, a, I, or i {4.2.2.5}.[6] If *type* is omitted or null, then "1" is assumed. If *text-indent* is non-null, it

5. Values that specify indentation must be *unscaled* and are treated as "character positions," i.e., as the number of *ens*.
6. Note that the "0001" format is *not* permitted.

is used as the number of spaces from the current indent to the text, i.e., it is used instead of *Li* for this list only. If *text-indent* is null, then the value of *Li* will be used.

If the third argument is given, a blank line (½ a vertical space) will *not* separate the items in the list. A blank line (½ a vertical space) will occur before the first item, however.

5.3.3.2 Bullet List

.BL [text-indent] [1]

.BL begins a bullet list, in which each item is marked by a bullet (•) followed by one space. If *text-indent* is non-null, it overrides the default indentation—the amount of paragraph indentation as given in the register *Pi* {4.1}.[7]

If a second argument is specified, no blank lines will separate the items in the list.

5.3.3.3 Dash List

.DL [text-indent] [1]

.DL is identical to .BL, except that a dash is used instead of a bullet.

5.3.3.4 Marked List

.ML mark [text-indent] [1]

.ML is much like .BL and .DL, but expects the user to specify an arbitrary mark, which may consist of more than a single character. Text is indented *text-indent* spaces if the second argument is not null; otherwise, the text is indented one more space than the width of

7. So that, in the default case, the text of bullet and dash lists lines up with the first line of indented paragraphs.

mark. If the third argument is specified, no blank lines will separate the items in the list.

CAUTION

The *mark* must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified {3.3}.

5.3.3.5 Reference List

.RL [text-indent] [1]

A .RL call begins an automatically-numbered list in which the numbers are enclosed by square brackets ([ ]). *Text-indent* may be supplied, as for .AL. If omitted or null, it is assumed to be 6, a convenient value for lists numbered up to 99. If the second argument is specified, no blank lines will separate the items in the list.

5.3.3.6 Variable-Item List

.VL text-indent [mark-indent] [1]

When a list begins with a .VL, there is effectively no *current mark;* it is expected that each .LI will provide its own mark. This form is typically used to display definitions of terms or phrases. *Mark-indent* gives the number of spaces from the current indent to the beginning of the *mark,* and it defaults to 0 if omitted or null. *Text-indent* gives the distance from the current indent to the beginning of the text. If the third argument is specified, no blank lines will separate the items in the list. Here is an example of .VL usage:

```
.tr ~
.VL 20 2
.LI mark~1
Here is a description of mark 1;
``mark 1'' of the .LI line contains a tilde translated
to an unpaddable space in order
to avoid extra spaces between
``mark'' and ``1'' {3.3}.
.LI second~mark
This is the second mark, also using a tilde translated
to an unpaddable space.
.LI third~mark~longer~than~indent:
This item shows the effect of a long mark;
one space separates the mark
from the text.
.LI ~
This item effectively has no mark because the
tilde following the .LI is translated into a space.
.LE
```

yields:

mark 1
: Here is a description of mark 1; "mark 1" of the .LI line contains a tilde translated to an unpaddable space in order to avoid extra spaces between "mark" and "1" {3.3}.

second mark
: This is the second mark, also using a tilde translated to an unpaddable space.

third mark longer than indent:
: This item shows the effect of a long mark; one space separates the mark from the text.

  This item effectively has no mark because the tilde following the .LI is translated into a space.

The tilde argument on the last .LI above is required; otherwise a *hanging indent* would have been produced. A *hanging indent* is produced by using .VL and calling .LI with no arguments or with a null first argument. For example:

```
.VL 10
.LI
Here is some text to show a hanging indent.
The first line of text is at the left margin.
The second is indented 10 spaces.
.LE
```

yields:

Here is some text to show a hanging indent. The first line of text is at the left margin. The second is indented 10 spaces.

CAUTION

The *mark* must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified {3.3}.

5.4 List-Begin Macro and Customized Lists

.LB text-indent mark-indent pad type [mark] [LI-space] [LB-space]

The list-initialization macros described above suffice for almost all cases. However, if necessary, one may obtain more control over the layout of lists by using the basic list-begin macro .LB, which is also used by all the other list-initialization macros. Its arguments are as follows:

Text-indent gives the number of spaces that the text is to be indented from the current indent. Normally, this value is taken from the register *Li* for automatic lists and from the register *Pi* for bullet and dash lists.

The combination of *mark-indent* and *pad* determines the placement of the mark. The mark is placed within an area (called *mark area*) that starts *mark-indent* spaces to the right of the current indent, and ends where the text begins (i.e., ends *text-indent* spaces to the right of the current indent).[8] Within the mark area, the mark is *left-justified* if

pad is 0. If *pad* is greater than 0, say *n*, then *n* blanks are appended to the mark; the *mark-indent* value is ignored. The resulting string immediately precedes the text. That is, the mark is effectively *right-justified pad* spaces immediately to the left of the text.

Type and *mark* interact to control the type of marking used. If *type* is 0, simple marking is performed using the mark character(s) found in the *mark* argument. If *type* is greater than 0, automatic numbering or alphabetizing is done, and *mark* is then interpreted as the first item in the sequence to be used for numbering or alphabetizing, i.e., it is chosen from the set (1, A, a, I, i) as in {5.3.3.1}. That is:

Type	*Mark*	*Result*
0	omitted	hanging indent
0	*string*	*string* is the mark
>0	omitted	arabic numbering
>0	one of: 1, A, a, I, i	automatic numbering or alphabetic sequencing

Each non-zero value of *type* from 1 to 6 selects a different way of displaying the marks. The following table shows the output appearance for each value of *type:*

Type	*Appearance*
1	x.
2	x)
3	(x)
4	[x]
5	<x>
6	{x}

where *x* is the generated number or letter.

8. The *mark-indent* argument is typically 0.

CAUTION

The *mark* must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified {3.3}.

LI-space gives the number of blank lines (halves of a vertical space) that should be output by each .LI macro in the list. If omitted, *LI-space* defaults to 1; the value 0 can be used to obtain compact lists. If *LI-space* is greater than 0, the .LI macro issues a .ne request for two lines just before printing the mark.

LB-space, the number of blank lines (½ a vertical space) to be output by .LB itself, defaults to 0 if omitted.

There are three reasonable combinations of *LI-space* and *LB-space*. The normal case is to set *LI-space* to 1 and *LB-space* to 0, yielding one blank line *before* each item in the list; such a list is usually terminated with a ".LE 1" to end the list with a blank line. In the second case, for a more compact list, set *LI-space* to 0 and *LB-space* to 1, and, again, use ".LE 1" at the end of the list. The result is a list with one blank line before and after it. If you set both *LI-space* and *LB-space* to 0, and use ".LE" to end the list, a list without *any* blank lines will result.

Appendix A shows how one can build upon the supplied list macros to obtain other kinds of lists.

6. Memorandum and Released Paper Styles

One use of MM is for the preparation of memoranda and released papers, which have special requirements for the first page and for the cover sheet. The information needed for the memorandum or released paper (title, author, date, case numbers, etc.) is entered in the same way for *both* styles; an argument to one macro indicates which style is being used. The following sections describe the macros used to provide this data. The required order is shown in {6.9}.

If neither the memorandum nor released-paper style is desired, the macros described below should be omitted from the input text. If these macros are omitted, the first page will simply have the page header {9} followed by the body of the document.

6.1 Title

```
.TL [charging-case] [filing-case]
one or more lines of title text
```

The arguments to the .TL macro are the charging case number(s) and filing case number(s).[9] The title of the memorandum or paper follows the .TL macro and is processed in fill mode {3.1}. Multiple charging case numbers are entered as "sub-arguments" by separating each from the previous with a comma and a space, and enclosing the *entire* argument within double quotes. Multiple filing case numbers are entered similarly. For example:

```
.TL "12345, 67890" 987654321
On the Construction of a Table of All Even Prime Numbers
```

The .br request may be used to break the title on output into several lines as follows:

```
.TL 12345
First Title Line
.br
\!.br
Second Title Line
```

On output, the title appears after the word "subject" in the memorandum style. In the released-paper style, the title is centered and bold.

If only a charging case number or only a filing case number is given, then it will be separated from the title in the memorandum style by a dash and will appear on the same line as the title. If both case numbers are given and are the same, then "Charging and Filing Case" followed by the number will appear on a line following the title. If the two case numbers are different, then separate lines for "Charging Case" and "File Case" will appear after the title.

9. The "charging case" is the case number to which time was charged for the development of the project described in the memorandum. The "filing case" is a number under which the memorandum is to be filed.

6.2 Author(s)

.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]
.AT [title] ...

The .AU macro receives as arguments information that describes an author. If any argument contains blanks, it must be enclosed within double quotes. The first six arguments must appear in the order given (a separate .AU macro is required for each author).

The .AT macro is used to specify the author's title. Up to nine arguments may be given. Each will appear in the Signature Block for memorandum style {6.11.1} on a separate line following the signer's name. The .AT must immediately follow the .AU for the given author. For example:

```
.AU "J. J. Jones" JJJ PY 9876 5432 1Z-234
.AT Director "Materials Research Laboratory"
```

In the "from" portion in the memorandum style, the author's name is followed by location and department number on one line and by room number and extension number on the next. The "x" for the extension is added automatically. The printing of the location, department number, extension number, and room number may be suppressed on the first page of a memorandum by setting the register *Au* to 0; the default value for *Au* is 1. Arguments 7 through 9 of the .AU macro, if present, will follow this "normal" author information in the "from" portion, each on a separate line. These last three arguments may be used for organizational numbering schemes, etc. For example:

```
.AU "S. P. Lename" SPL IH 9988 7766 5H-444 9876-543210.01MF
```

The name, initials, location, and department are also used in the Signature Block {6.11.1}. The author information in the "from" portion, as well as the names and initials in the Signature Block will appear in the same order as the .AU macros.

The names of the authors in the released-paper style are centered below the title. After the name of the last author, "Bell Laboratories" and the location are centered. For the case of authors from different locations, see {6.8}.

6.3 TM Number(s)

.TM [number] ...

If the memorandum is a Technical Memorandum, the TM numbers are supplied via the .TM macro. Up to nine numbers may be specified. Example:

.TM 7654321 77777777

This macro call is ignored in the released-paper and external-letter styles {6.6}.

6.4 Abstract

.AS [arg] [indent]
text of the abstract
.AE

The .AS (abstract start) and .AE (abstract end) macros bracket the (optional) abstract. Abstracts are printed on page 1 of a document and/or on its cover sheet.[10]

In a released paper (first argument of the .MT macro is 4; see {6.6}) and in a Technical Memorandum, if the first argument of .AS is 0, the abstract will be printed on page 1 *and* on the cover sheet (if any); if the first argument of .AS is 1, the abstract will appear *only* on the cover sheet (if any).

In Memoranda for File and in all other documents (other than external letters), if the first argument of .AS is 0, the abstract will appear on page 1 and there will be no cover sheet printed; if the first argument of .AS is 2, the abstract will appear *only* on the cover sheet, which will be produced automatically in this case (i.e., *without* invoking the .CS macro). It is *not* possible to get either an abstract or a

10. There are three styles of cover sheet: released paper, Technical Memorandum, and Memorandum for File {10.2}; the last one of these is also used for Engineer's Notes, Memoranda for Record, etc. Cover sheets for released papers and Technical Memoranda are obtained by invoking the .CS macro {10.2}.

cover sheet with an external letter (first argument of the .MT macro is 5).

Notations {6.11.2} such as a "copy to" list are allowed on Memorandum for File cover sheets; the .NS and .NE macros must appear after the .AS 2 and .AE. Headings {4.2, 4.3} and displays {7} are *not* permitted within an abstract.

The abstract is printed with ordinary text margins; an indentation to be used for both margins can be specified as the second argument of .AS.[11]

6.5 Other Keywords

.OK [keyword] ...

Topical keywords should be specified on a Technical Memorandum cover sheet. Up to nine such keywords or keyword phrases may be specified as arguments to the .OK macro; if any keyword contains spaces, it must be enclosed within double quotes.

6.6 Memorandum Types

.MT [type] [addressee]

The .MT macro controls the format of the top part of the first page of a memorandum or of a released paper, as well as the format of the cover sheets. Legal codes for *type* and the corresponding values are:

11. Values that specify indentation must be *unscaled* and are treated as "character positions," i.e., as the number of *ens*.

Code	*Value*
" "	no memorandum type printed
0	no memorandum type printed
none	MEMORANDUM FOR FILE
1	MEMORANDUM FOR FILE
2	PROGRAMMER'S NOTES
3	ENGINEER'S NOTES
4	released-paper style
5	external-letter style
"*string*"	*string*

If *type* indicates a memorandum style, then *value* will be printed after the last line of author information. If *type* is longer than one character, then the string, itself, will be printed. For example:

```
.MT "Technical Note #5"
```

A simple letter is produced by calling .MT with a null (but *not* omitted!) or zero argument.

The second argument to .MT is the name of the addressee of a letter; if present, that name and the page number replace the normal page header on the second and following pages of a letter:

```
.MT 1 "John Jones"
```

produces

```
John Jones – 2
```

This second argument may *not* be used for this purpose if the first argument is 4 (i.e., for the released-paper style) as explained in {6.8}.

In the external-letter style (.MT 5), only the title (without the word "subject:") and the date are printed in the upper left and right corners, respectively, on the first page. It is expected that preprinted stationery will be used, providing the author's company logo and address.

6.7 Date and Format Changes

6.7.1 Changing the Date

By default, the current date appears in the "date" part of a memorandum. This can be overridden by using:

.ND new-date

The .ND macro alters the value of the string *DT*, which is initially set to the current date.

6.7.2 Alternate First-Page Format.

One can specify that the words "subject," "date," and "from" (in the memorandum style) be omitted and that an alternate company name be used:

.AF [company-name]

If an argument is given, it replaces "Bell Laboratories", without affecting the other headings. If the argument is *null*, "Bell Laboratories" is suppressed; in this case, extra blank lines are inserted to allow room for stamping the document with a Bell System logo or a Bell Laboratories stamp. .AF with *no* argument suppresses "Bell Laboratories" and the "Subject/Date/From" *headings*, thus allowing output on preprinted stationery. The use of .AF with no arguments is equivalent to the use of –rA1 {2.4}, except that the latter *must* be used if it is necessary to change the line length and/or page offset (which default to 5.8i and 1i, respectively, for preprinted forms). The command line options –rOk and –rWk {2.4} are *not* effective with .AF. The only .AF use appropriate for *troff* is to specify a replacement for "Bell Laboratories".

The command line option –rE*n* {2.4} controls the font of the "Subject/Date/From" block.

6.8 Released-Paper Style

The released-paper style is obtained by specifying:

```
.MT 4 [1]
```

This results in a centered, bold title followed by centered names of authors. The location of the last author is used as the location following "Bell Laboratories" (unless .AF {6.7.2} specifies a different company). If the optional second argument to .MT 4 is given, then the name of each author is followed by the respective company name and location.

Information necessary for the memorandum style but not for the released-paper style is ignored.

If the released-paper style is utilized, most BTL location codes[12] are defined as strings that are the addresses of the corresponding BTL locations. These codes are needed only until the .MT macro is invoked. Thus, *following* the .MT macro, the user may re-use these string names. In addition, the macros described in {6.11} and their associated lines of input are ignored when the released-paper style is specified.

Authors from non-BTL locations may include their affiliations in the released-paper style by specifying the appropriate .AF and defining a string (with a 2 character name such as XX) for the address *before* each .AU. For example:

```
.TL
A Learned Treatise
.AF "Getem Inc."
.ds XX "22 Maple Avenue, Sometown 09999"
.AU "F. Swatter" "" XX
.AF "Bell Laboratories"
.AU "Sam P. Lename" "" CB
.MT 4 1
```

12. Currently, the complete list is: AK, AL, ALF, CB, CH, CP, DR, FJ, HL, HO, HOH, HP, IH, IN, INH, IW, MH, MV, PY, RD, RR, WB, WH, and WV.

6.9 Order of Invocation of "Beginning" Macros

The macros described in {6.1-6.7}, *if present*, must be given in the following order:

```
.ND new-date
.TL [charging-case] [filing-case]
one or more lines of text
.AF [company-name]
.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]
.AT [title] ...
.TM [number] ...
.AS [arg] [indent]
one or more lines of text
.AE
.NS [arg]
one or more lines of text
.NE
.OK [keyword] ...
.MT [type] [addressee]
```

The only *required* macros for a memorandum or a released paper are .TL, .AU, and .MT; all the others (and their associated input lines) may be omitted if the features they provide are not needed. Once .MT has been invoked, *none* of the above macros (except .NS and .NE) can be re-invoked because they are removed from the table of defined macros to save space.

6.10 Example

The input text for this manual begins as follows:

```
.TL
MM\*(EMMemorandum Macros
.AU "D. W. Smith" DWS PY
.AU "J. R. Mashey" JRM PY
.AU "E. C. Pariser (January 1980 Revision)" ECP PY
.AU "N. W. Smith (June 1980 Revision)" NWS PY
.MT 4
```

Appendix C shows the input and both *nroff* and *troff* outputs for a simple letter.

6.11 Macros for the End of a Memorandum

At the end of a memorandum (but *not* of a released paper), the signatures of the authors and a list of notations[13] can be requested. The following macros and their input are ignored if the released-paper style is selected.

6.11.1 Signature Block

```
.FC [closing]
.SG [arg] [1]
```

.FC prints "Yours very truly," as a formal closing. It must be given before the .SG which prints the signer's name. A different closing may be specified as an argument to .FC.

.SG prints the author name(s) after the formal closing, if any. Each name begins at the center of the page. Three blank lines are left above each name for the actual signature. If no arguments are given, the line of reference data[14] will *not* appear. A non-null first argument is treated as the typist's initials, and is appended to the reference data. Supply a null first argument to print the reference data with neither the typist's initials nor the preceding hyphen.

If there are several authors and if the second argument is given, then the reference data is placed on the same line as the name of the first, rather than last, author.

The reference data contains only the location and department number of the first author. Thus, if there are authors from different departments and/or from different locations, the reference data should be

13. See the *BTL Office Guide*[9], pp. 1.12-16.
14. The following information is known as reference data: location code, department number, author's initials, and typist's initials, all separated by hyphens. See the *BTL Office Guide*[9], page 1.11.

supplied manually after the invocation (without arguments) of the .SG macro. For example:

```
.SG
.rs
.sp -1v
PY/MH-9876/5432-JJJ/SPL-cen
```

6.11.2 "Copy to" and Other Notations

```
.NS [arg]
zero or more lines of the notation
.NE
```

After the signature and reference data, many types of notations may follow, such as a list of attachments or "copy to" lists. The various notations are obtained through the .NS macro, which provides for the proper spacing and for breaking the notations across pages, if necessary.

The codes for *arg* and the corresponding notations are:

Code	*Notations*
none	Copy to
" "	Copy to
0	Copy to
1	Copy (with att.) to
2	Copy (without att.) to
3	Att.
4	Atts.
5	Enc.
6	Encs.
7	Under Separate Cover
8	Letter to
9	Memorandum to
"*string*"	Copy (*string*) to

If *arg* consists of more than one character, it is placed within parentheses between the words "Copy" and "to." For example:

```
.NS "with att. 1 only"
```

will generate "Copy (with att. 1 only) to" as the notation. More than one notation may be specified before the .NE occurs, because a .NS macro terminates the preceding notation, if any. For example:

```
.NS 4
Attachment 1-List of register names
Attachment 2-List of string and macro names
.NS 1
J. J. Jones
.NS 2
S. P. Lename
G. H. Hurtz
.NE
```

would be formatted as:

Atts.
Attachment 1-List of register names
Attachment 2-List of string and macro names

Copy (with att.) to
J. J. Jones

Copy (without att.) to
S. P. Lename
G. H. Hurtz

The .NS and .NE macros may also be used at the beginning following .AS 2 and .AE to place the notation list on the Memorandum for File cover sheet {6.4}. If notations are given at the beginning without .AS 2, they will be saved and output at the end of the document.

6.11.3 Approval Signature Line

```
.AV approver's-name
```

The .AV macro may be used after the last notation block to automatically generate a line with spaces for the approval signature and date. For example,

```
.AV "Jane Doe"
```

produces:

APPROVED:

Jane Doe _____ Date _____

6.12 Forcing a One-Page Letter

At times, one would like just a bit more space on the page, forcing the signature or items within notations onto the bottom of the page, so that the letter or memo is just one page in length. This can be accomplished by increasing the page length through the –rL*n* option, e.g. –rL90. This has the effect of making the formatter believe that the page is 90 lines long and therefore giving it more room than usual to place the signature or the notations. This will *only* work for a *single-page* letter or memo.

7. Displays

Displays are blocks of text that are to be kept together—not split across pages. MM provides two styles of displays:[15] a *static* (.DS) style and a *floating* (.DF) style. In the *static* style, the display appears in the same relative position in the output text as it does in the input text; this may result in extra white space at the bottom of the page if the display is too big to fit there. In the *floating* style, the display "floats" through the input text to the top of the next page if there is not enough room for it on the current page; thus the input text that *follows* a floating display may *precede* it in the output text. A queue of floating displays is maintained so that their relative order is not disturbed.

By default, a display is processed in no-fill mode, with single-spacing, and is *not* indented from the existing margins. The user can specify indentation or centering, as well as fill-mode processing.

15. Displays are processed in an environment that is different from that of the body of the text (see the .ev request[2]).

Displays and footnotes {8} may *never* be nested, in any combination whatsoever. Although lists {5} and paragraphs {4.1} are permitted, no headings (.H or .HU) {4.2, 4.3} can occur within displays or footnotes.

7.1 Static Displays

.DS [format] [fill] [rindent]
one or more lines of text
.DE

A static display is started by the .DS macro and terminated by the .DE macro. With no arguments, .DS will accept the lines of text exactly as they are typed (no-fill mode) and will *not* indent them from the prevailing left margin indentation or from the right margin. The *rindent* argument is the number of characters[16] that the line length should be decreased, i.e., an indentation from the right margin.

The *format* argument to .DS is an integer or letter used to control the left margin indentation and centering with the following meanings:

Format	*Meaning*
" "	no indent
0 or L	no indent
1 or I	indent by standard amount
2 or C	center each line
3 or CB	center as a block

The *fill* argument is also an integer or letter and can have the following meanings:

Fill	*Meaning*
" "	no-fill mode
0 or N	no-fill mode
1 or F	fill mode

Omitted arguments are taken to be zero.

16. This number must be unscaled in *nroff* and is treated as *ens*. It may be scaled in *troff* or else defaults to *ems*.

The standard amount of indentation is taken from the register *Si*, which is initially 5. Thus, by default, the text of an indented display aligns with the first line of indented paragraphs, whose indent is contained in the *Pi* register {4.1}. Even though their initial values are the same, these two registers are independent of one another.

The display format value 3 (CB) centers the *entire display* as a block (as opposed to .DS 2 and .DF 2, which center each line *individually*). That is, all the collected lines are left-justified, and then the display is centered, based on the width of the longest line. This format *must* be used in order for the *eqn/neqn* "mark" and "lineup" feature to work with centered equations (see section 7.4 below).

By default, a blank line (½ a vertical space) is placed before and after static and floating displays. These blank lines before and after *static* displays can be inhibited by setting the register *Ds* to 0.

The following example shows the usage of all three arguments for displays. This block of text will be filled and indented 5 spaces from both the left and the right margins (i.e., centered).

```
.DS I F 5
"We the people of the United States, in order to form
a more perfect union, establish justice, ensure domes-
tic tranquility, provide for the common defense, and
secure the blessings of liberty to ourselves and our pos-
terity, do ordain and establish this Constitution to the
United States of America."
.DE
```

7.2 Floating Displays

```
.DF [format] [fill] [rindent]
one or more lines of text
.DE
```

A floating display is started by the .DF macro and terminated by the .DE macro. The arguments have the same meanings as for .DS {7.1}, except that, for floating displays, indent, no indent, and centering are always calculated with respect to the initial left margin, because the prevailing indent may change between the time when the formatter first reads the floating display and the time that the display is printed. One blank line (½ a vertical space) *always* occurs both before and after a floating display.

The user may exercise great control over the output positioning of floating displays through the use of two number registers, *De* and *Df*. When a floating display is encountered by *nroff* or *troff*, it is processed and placed onto a queue of displays waiting to be output. Displays are always removed from the queue and printed in the order that they were entered on the queue, which is the order that they appeared in the input file. If a new floating display is encountered and the queue of displays is empty, then the new display is a candidate for immediate output on the current page. Immediate output is governed by the size of the display and the setting of the *Df* register (see below). The *De* register (see below) controls whether or not text will appear on the current page after a floating display has been produced.

As long as the queue contains one or more displays, new displays will be automatically entered there, rather than being output. When a new page is started (or the top of the second column when in two-column mode) the next display from the queue becomes a candidate for output if the *Df* register has specified "top-of-page" output. When a display is output it is also removed from the queue.

When the end of a section (when using section-page numbering) or the end of a document is reached, all displays are automatically removed from the queue and output. This will occur before a .SG, .CS, or .TC is processed.

A display is said to "fit on the current page" if there is enough room to contain the entire display on the page, or if the display is longer than one page in length and less than half of the current page has been used. Also note that a wide (full page width) display will never fit in the second column of a two-column document.

The registers, their settings, and their effects are as follows:

De	*Action*
0	DEFAULT: No special action occurs.
1	A page eject will *always* follow the output of each floating display, so only one floating display will appear on a page and no text will follow it.
	NOTE: For any other value, the action performed is the same as for the value 1.

Df	*Action*
0	Floating displays will not be output until end of section (when section-page numbering) or end of document.
1	Output the new floating display on the current page if there is room, otherwise hold it until the end of the section or document.
2	Output exactly one floating display from the queue at the top of a new page or column (when in two-column mode).
3	Output one floating display on current page if there is room. Output exactly one floating display at the top of a new page or column.
4	Output as many displays as will fit (at least one), starting at the top of a new page or column. Note that if register *De* is set to 1, each display will be followed by a page eject, causing a new top of page to be reached where at least one more display will be output. (This also applies to value 5, below.)
5	DEFAULT: Output a new floating display on the current page if there is room. Output at least one, but as many displays as will fit starting at the top of a new page or column.
	NOTE: For any value greater than 5, the action performed is the same as for the value 5.

The .WC macro {12.4} may also be used to control handling of displays in double-column mode and to control the break in the text before floating displays.

7.3 Tables

.TS [H]
global options;
column descriptors.
title lines
[.TH [N]]
data within the table.
.TE

The .TS (table start) and .TE (table end) macros make possible the use of the *tbl*(1) processor[7]. They are used to delimit the text to be examined by *tbl*(1) as well as to set proper spacing around the table. The display function and the *tbl*(1) delimiting function are independent of one another, however, so in order to permit one to keep

together blocks that contain any mixture of tables, equations, filled and unfilled text, and caption lines the .TS/.TE block should be enclosed within a display (.DS/.DE). Floating tables may be enclosed inside floating displays (.DF/.DE).

The macros .TS and .TE also permit the processing of tables that extend over several pages. If a table heading is needed for each page of a multi-page table, specify the argument "H" to the .TS macro as above. Following the options and format information, the table heading is typed on as many lines as required and followed by the .TH macro. The .TH macro *must* occur when ".TS H" is used. Note that this is *not* a feature of *tbl*(1), but of the macro definitions provided by MM.

The table header macro .TH may take as an argument the letter N. This argument causes the table header to be printed only if it is the first table header on the page. This option is used when it is necessary to build long tables from smaller .TS H/.TE segments. For example:

```
.TS H
global options;
column descriptors.
Title lines
.TH
data
.TE
.TS H
global options;
column descriptors.
Title lines
.TH N
data
.TE
```

will cause the table heading to appear at the top of the first table segment, and no heading to appear at the top of the second segment when both appear on the same page. However, the heading will still appear at the top of each page that the table continues onto. This feature is used when a single table must be broken into segments because of table complexity (for example, too many blocks of filled text). If each segment had its own .TS H/.TH sequence, each segment would have its own header. However, if each table segment

after the first uses .TS H/.TH N then the table header will only appear at the beginning of the table and the top of each new page or column that the table continues onto.

For *nroff*, the –e option (–E for *mm*(1) {2.1}) may be used for terminals, such as the 450, that are capable of finer printing resolution. This will cause better alignment of features such as the lines forming the corner of a box. Note that –e is not effective with *col*(1).

7.4 Equations

```
.DS
.EQ [label]
equation(s)
.EN
.DE
```

The equation setters *eqn*(1) and *neqn*[6] expect to use the .EQ (equation start) and .EN (equation end) macros as delimiters in the same way that *tbl*(1) uses .TS and .TE; however, .EQ and .EN must occur inside a .DS/.DE pair.

NOTE

> There is an exception to this rule: if .EQ and .EN are used only to specify the delimiters for in-line equations or to specify *eqn*/*neqn* "defines," .DS and .DE must *not* be used; otherwise extra blank lines will appear in the output.

The .EQ macro takes an argument that will be used as a label for the equation. By default, the label will appear at the right margin in the "vertical center" of the general equation. The *Eq* register may be set to 1 to change the labeling to the left margin.

The equation will be centered for centered displays; otherwise the equation will be adjusted to the opposite margin from the label.

7.5 Figure, Table, Equation, and Exhibit Captions

.FG [title] [override] [flag]
.TB [title] [override] [flag]
.EC [title] [override] [flag]
.EX [title] [override] [flag]

The .FG (Figure Title), .TB (Table Title), .EC (Equation Caption) and .EX (Exhibit Caption) macros are normally used inside .DS/.DE pairs to automatically number and title figures, tables, and equations. They use registers *Fg*, *Tb*, *Ec*, and *Ex*, respectively (see ⟨2.4⟩ on –rN5 to reset counters in sections). As an example, the call:

.FG "This is an illustration"

yields:

Figure 1. This is an illustration

.TB replaces "Figure" by "TABLE"; .EC replaces "Figure" by "Equation", and .EX replaces "Figure" by "Exhibit". Output is centered if it can fit on a single line; otherwise, all lines but the first are indented to line up with the first character of the title. The format of the numbers may be changed using the .af request of the formatter. The format of the caption may be changed from "Figure 1. Title" to "Figure 1 - Title" by setting the *Of* register to 1.

The *override* string is used to modify the normal numbering. If *flag* is omitted or 0, *override* is used as a prefix to the number; if *flag* is 1, *override* is used as a suffix; and if *flag* is 2, *override* replaces the number. If –rN5 ⟨2.4⟩ is given, "section-figure" numbering is set automatically and user-specified *override* string is ignored.

As a matter of style, table headings are usually placed ahead of the text of the tables, while figure, equation, and exhibit captions usually occur after the corresponding figures and equations.

7.6 List of Figures, Tables, Equations, and Exhibits

A List of Figures, List of Tables, List of Exhibits, and List of Equations may be obtained. They will be printed after the Table of Contents is printed if the number registers *Lf*, *Lt*, *Lx*, and *Le* (respectively) are set to 1. *Lf*, *Lt*, and *Lx* are 1 by default; *Le* is 0 by default.

The titles of these Lists may be changed by redefining the following strings which are shown here with their default values:

```
.ds Lf LIST OF FIGURES
.ds Lt LIST OF TABLES
.ds Lx LIST OF EXHIBITS
.ds Le LIST OF EQUATIONS
```

8. Footnotes

There are two macros that delimit the text of footnotes,[17] a string used to automatically number the footnotes, and a macro that specifies the style of the footnote text.

8.1 Automatic Numbering of Footnotes

Footnotes may be automatically numbered by typing the three characters "*F" (i.e., invoking the string *F*) immediately after the text to be footnoted, without any intervening spaces. This will place the next sequential footnote number (in a smaller point size) a half-line above the text to be footnoted.

17. Footnotes are processed in an environment that is different from that of the body of the text (see the .ev request[2]).

8.2 Delimiting Footnote Text

There are two macros that delimit the text of each footnote:

```
.FS [label]
one or more lines of footnote text
.FE
```

The .FS (footnote start) marks the beginning of the text of the footnote, and the .FE marks its end. The *label* on the .FS, if present, will be used to mark the footnote text. Otherwise, the number retrieved from the string F will be used. Note that automatically-numbered and user-labeled footnotes may be intermixed. If a footnote is labeled (.FS *label*), the text to be footnoted *must* be followed by *label*, rather than by "*F". The text between .FS and .FE is processed in fill mode. Another .FS, a .DS, or a .DF are *not* permitted between the .FS and .FE macros. Automatically numbered footnotes may not be used for information, such as the title and abstract, to be placed on the cover sheet, but labeled footnotes are allowed. Similarly, only labeled footnotes may be used with tables {7.3}. Examples:

1. Automatically-numbered footnote:

```
This is the line containing the word\*F
.FS
This is the text of the footnote.
.FE
to be footnoted.
```

2. Labelled footnote:

```
This is a labeled*
.FS *
The footnote is labeled with an asterisk.
.FE
footnote.
```

The text of the footnote (enclosed within the .FS/.FE pair) should *immediately* follow the word to be footnoted in the input text, so that "*F" or *label* occurs at the end of a line of input and the next line is the .FS macro call. It is also good practice to append a unpaddable space {3.3} to "*F" or *label* when they follow an end-of-sentence punctuation mark (i.e., period, question mark, exclamation point).

Appendix B illustrates the various available footnote styles as well as numbered and labeled footnotes.

8.3 Format of Footnote Text

.FD [arg] [1]

Within the footnote text, the user can control the formatting style by specifying text hyphenation, right margin justification, and text indentation, as well as left- or right-justification of the label when text indenting is used. The .FD macro is invoked to select the appropriate style. The first argument is a number from the left column of the following table. The formatting style for each number is given by the remaining four columns. For further explanation of the first two of these columns, see the definitions of the .ad, .hy, .na, and .nh requests[2].

arg	*Hyphenation*	*Adjust*	*Text Indent*	*Label Justification*
0	.nh	.ad	text indent	label left justified
1	.hy	.ad	"	"
2	.nh	.na	"	"
3	.hy	.na	"	"
4	.nh	.ad	no text indent	"
5	.hy	.ad	"	"
6	.nh	.na	"	"
7	.hy	.na	"	"
8	.nh	.ad	text indent	label right justified
9	.hy	.ad	"	"
10	.nh	.na	"	"
11	.hy	.na	"	"

If the first argument to .FD is out of range, the effect is as if .FD 0 were specified. If the first argument is omitted or null, the effect is equivalent to .FD 10 in *nroff* and to .FD 0 in *troff;* these are also the respective initial defaults.

If a second argument is specified, then whenever a first-level heading is encountered, automatically-numbered footnotes begin again with 1. This is most useful with the "section-page" page numbering scheme. As an example, the input line:

.FD " " 1

maintains the default formatting style and causes footnotes to be numbered afresh after each first-level heading.

For long footnotes that continue onto the following page, it is possible that, if hyphenation is permitted, the last line of the footnote on the current page will be hyphenated. Except for this case (over which the user has control by specifying an *even* argument to .FD), hyphenation across pages is inhibited by MM.

Footnotes are separated from the body of the text by a short rule. Footnotes that continue to the next page are separated from the body of the text by a full-width rule. In *troff*, footnotes are set in type that is two points smaller than the point size used in the body of the text.

8.4 Spacing between Footnote Entries

Normally, one blank line (a three-point vertical space) separates the footnotes when more than one occurs on a page. To change this spacing, set the register *Fs* to the desired value. For example:

```
.nr Fs 2
```

will cause two blank lines (a six-point vertical space) to occur between footnotes.

9. Page Headers and Footers

Text that occurs at the top of each page is known as the *page header*. Text printed at the bottom of each page is called the *page footer*. There can be up to three lines of text associated with the header: every page, even page only, and odd page only. Thus the page header may have up to two lines of text: the line that occurs at the top of every page and the line for the even- or odd-numbered page. The same is true for the page footer.

This section first describes the default appearance of page headers and page footers, and then the ways of changing them. We use the term *header (not* qualified by *even* or *odd)* to mean the line of the page header that occurs on every page, and similarly for the term *footer*.

9.1 Default Headers and Footers

By default, each page has a centered page number as the header {9.2}. There is no default footer and no even/odd default headers or footers, except as specified in {9.9}.

In a memorandum or a released paper, the page header on the first page is automatically suppressed *provided* a break does *not* occur before .MT is called. The macros and text of {6.9} and of {9} as well as .nr and .ds requests do *not* cause a break and are permitted before the .MT macro call.

9.2 Page Header

.PH [arg]

For this and for the .EH, .OH, .PF, .EF, .OF macros, the argument is of the form:

" 'left-part'center-part'right-part' "

If it is inconvenient to use the apostrophe (') as the delimiter (i.e., because it occurs within one of the parts), it may be replaced *uniformly* by *any* other character. On output, the parts are left-justified, centered, and right-justified, respectively. See {9.11} for examples.

The .PH macro specifies the header that is to appear at the top of every page. The initial value (as stated in {9.1}) is the default centered page number enclosed by hyphens. The page number contained in the *P* register is an Arabic number. The format of the number may be changed by the .af request.

If *debug mode* is set using the flag –rD1 on the command line {2.4}, additional information, printed at the top left of each page, is included in the default header. This consists of the SCCS[10] Release and Level of MM (thus identifying the current version {12.3}), followed by the current line number within the current input file.

9.3 Even-Page Header

.EH [arg]

The .EH macro supplies a line to be printed at the top of each even-numbered page, immediately *following* the header. The initial value is a blank line.

9.4 Odd-Page Header

.OH [arg]

This macro is the same as .EH, except that it applies to odd-numbered pages.

9.5 Page Footer

.PF [arg]

The .PF macro specifies the line that is to appear at the bottom of each page. Its initial value is a blank line. If the –rC*n* flag is specified on the command line {2.4}, the type of copy *follows* the footer on a separate line. In particular, if –rC3 or –rC4 (DRAFT) is specified, then, in addition, the footer is initialized to contain the date {6.7.1}, instead of being a blank line.

9.6 Even-Page Footer

.EF [arg]

The .EF macro supplies a line to be printed at the bottom of each even-numbered page, immediately *preceding* the footer. The initial value is a blank line.

9.7 Odd-Page Footer

.OF [arg]

This macro is the same as .EF, except that it applies to odd-numbered pages.

9.8 Footer on the First Page

By default, the footer is a blank line. If, in the input text, one specifies .PF and/or .OF before the end of the first page of the document, then these lines will appear at the bottom of the first page.

The header (whatever its contents) *replaces* the footer *on the first page only* if the –rN1 flag is specified on the command line {2.4}.

9.9 Default Header and Footer with "Section-Page" Numbering

Pages can be numbered sequentially within sections {4.5}. To obtain this numbering style, specify –rN3 or –rN5 on the command line. In this case, the default *footer* is a centered "section-page" number, e.g. 7-2, and the default page header is blank.

9.10 Use of Strings and Registers in Header and Footer Macros

String and register names may be placed in the arguments to the header and footer macros. If the value of the string or register is to be computed *when the respective header or footer is printed,* the invocation must be escaped by four (4) backslashes. This is because the string or register invocation will be processed three times:

- as the argument to the header or footer macro;
- in a formatting request within the header or footer macro;
- in a .tl request during header or footer processing.

For example, the page number register *P* must be escaped with four backslashes in order to specify a header in which the page number is to be printed at the right margin, e.g.:

```
.PH "'''Page \\\\nP'"
```

creates a right-justified header containing the word "Page" followed by the page number. Similarly, to specify a footer with the "section-page" style, one specifies (see {4.2.2.5} for meaning of *H1*):

```
.PF "'''- \\\\n(H1-\\\\nP -'"
```

As another example, suppose that the user arranges for the string *a]* to contain the current section heading which is to be printed at the bottom of each page. The .PF macro call would then be:

```
.PF "''\\\\*(a]''"
```

If only one or two backslashes were used, the footer would print a constant value for *a],* namely, its value when the .PF appeared in the input text.

9.11 Header and Footer Example

The following sequence specifies blank lines for the header and footer lines, page numbers on the outside edge of each page (i.e., top left margin of even pages and top right margin of odd pages), and "Revision 3" on the top inside margin of each page:

```
.PH " "
.PF " "
.EH "'\\\\nP''Revision 3'"
.OH "'Revision 3''\\\\nP'"
```

9.12 Generalized Top-of-Page Processing

NOTE

This section is intended only for users accustomed to writing formatter macros.

During header processing, MM invokes two user-definable macros. One, the .TP macro, is invoked in the environment (see .ev request[2]) of the header; the other, .PX, is a user-exit macro that is invoked (without arguments) when the normal environment has been restored, and with "no-space" mode already in effect.

The effective initial definition of .TP (after the first page of a document) is:

```
.de TP
'sp 3
.tl \\*(}t
.if e 'tl \\*(}e
.if o 'tl \\*(}o
'sp 2
..
```

The string *}t* contains the header, the string *}e* contains the even-page header, and the string *}o* contains the odd-page header, as defined by the .PH, .EH, and .OH macros, respectively. To obtain more specialized page titles, the user may redefine the .TP macro to cause any desired header processing {12.5}. Note that formatting done within the .TP macro is processed in an environment different from that of the body.

For example, to obtain a page header that includes three centered lines of data, say, a document's number, issue date, and revision date, one could define .TP as follows:

```
.de TP
.sp
.ce 3
777–888–999
Iss. 2, AUG 1977
Rev. 7, SEP 1977
.sp
..
```

The .PX macro may be used to provide text that is to appear at the top of each page after the normal header and that may have tab stops to align it with columns of text in the body of the document.

9.13 Generalized Bottom-of-Page Processing

.BS
zero or more lines of text
.BE

Lines of text that are specified between the .BS (bottom-block start) and .BE (bottom-block end) macros will be printed at the bottom of each page,[18] after the footnotes (if any), but before the page footer. This block of text is removed by specifying an empty block, i.e.:

.BS
.BE

9.14 Top and Bottom Margins

.VM [top] [bottom]

.VM (Vertical Margin) allows the user to specify extra space at the top and bottom of the page. This space precedes the page header and follows the page footer. .VM takes two unscaled arguments that are treated as v's. For example:

.VM 10 15

adds 10 blank lines to the default top of page margin, and 15 blank lines to the default bottom of page margin. Both arguments must be positive (default spacing at the top of the page may be decreased by redefining .TP).

18. The bottom block will appear on the table of contents pages and the cover sheet for Memorandum for File, but *not* on the Technical Memorandum or released-paper cover sheets.

9.15 Proprietary Markings

.PM [code]

.PM, for Proprietary Marking, appends to the page footer a PRIVATE, NOTICE, BELL LABORATORIES PROPRIETARY, or BELL LABORATORIES RESTRICTED disclaimer. The *code* may be:

Code	*Meaning*
none	turn off previous disclaimer, if any
P	PRIVATE
N	NOTICE
BP	BELL LABORATORIES PROPRIETARY
BR	BELL LABORATORIES RESTRICTED

The disclaimers are in a form approved for use by the Bell System.

9.16 Private Documents

.nr Pv value

The word "PRIVATE" may be printed centered and underlined on the second line of a document (preceding the page header). This is done by setting the *Pv* register:

Value	*Meaning*
0	do not print PRIVATE (default)
1	PRIVATE on first page only
2	PRIVATE on all pages

If *Pv* is 2, the user definable .TP may not be used because .TP is used by MM to print PRIVATE on all pages except the first page of a memorandum on which .TP is not invoked.

10. Table of Contents and Cover Sheet

The table of contents and the cover sheet for a document are produced by invoking the .TC and .CS macros, respectively.

NOTE

> This section will refer to cover sheets for Technical Memoranda and released papers only. The mechanism for producing a Memorandum for File cover sheet was discussed earlier {6.4}.

These macros should normally appear only once at the *end* of the document, after the Signature Block {6.11.1} and Notations {6.11.2} macros. They may occur in either order.

The table of contents is produced at the end of the document because the entire document must be processed before the table of contents can be generated. Similarly, the cover sheet is often not needed, and is therefore produced at the end.

10.1 Table of Contents

.TC [slevel] [spacing] [tlevel] [tab] [hd1] [hd2] [hd3] [hd4] [hd5]

The .TC macro generates a table of contents containing the headings that were saved for the table of contents as determined by the value of the *Cl* register {4.4}. The arguments to .TC control the spacing before each entry, the placement of the associated page number, and additional text on the first page of the table of contents before the word "CONTENTS."

Spacing before each entry is controlled by the first two arguments; headings whose level is less than or equal to *slevel* will have *spacing* blank lines (halves of a vertical space) before them. Both *slevel* and *spacing* default to 1. This means that first-level headings are preceded by one blank line (½ a vertical space). Note that *slevel* does *not* control what levels of heading have been saved; the saving of headings is the function of the *Cl* register {4.4}.

The third and fourth arguments control the placement of the page number for each heading. The page numbers can be justified at the right margin with either blanks or dots ("leaders") separating the heading text from the page number, or the page numbers can follow the heading text. For headings whose level is less than or equal to *tlevel* (default 2), the page numbers are justified at the right margin. In this case, the value of *tab* determines the character used to separate the heading text from the page number. If *tab* is 0 (the default value), dots (i.e., leaders) are used; if *tab* is greater than 0, spaces are used. For headings whose level is greater than *tlevel*, the page numbers are separated from the heading text by two spaces (i.e., they are "ragged right").

All additional arguments (e.g., *head1*, *head2*, etc.), if any, are horizontally centered on the page, and precede the actual table of contents itself.

If the .TC macro is invoked with at most four arguments, then the user-exit macro .TX is invoked (without arguments) before the word "CONTENTS" is printed; or the user-exit macro .TY is invoked and the word "CONTENTS" is *not* printed. By defining .TX or .TY and invoking .TC with at most four arguments, the user can specify what needs to be done at the top of the (first) page of the table of contents. For example, the following input:

```
.de TX
.ce 2
Special Application
Message Transmission
.sp 2
.in +10n
Approved: \l'3i'
.in
.sp
..
.TC
```

yields:

Special Application
Message Transmission

Approved: _____

CONTENTS

⋮

If this macro were defined as .TY rather than .TX, the word "CONTENTS" would not appear. Defining .TY as an empty macro will suppress "CONTENTS" with no replacement:

```
.de TY
..
```

By default, the first level headings will appear in the table of contents at the left margin. Subsequent levels will be aligned with the text of headings at the preceding level. These indentations may be changed by defining the *Ci* string which takes a maximum of seven arguments corresponding to the heading levels. It must be given at least as many arguments as are set by the *Cl* register {4.4}. The arguments must be scaled. For example, with *Cl*=5,

```
.ds Ci .25i .5i .75i 1i 1i
```

or

```
.ds Ci 0 2n 4n 6n 8n
```

Two other registers are available to modify the format of the table of contents, *Oc* and *Cp*. By default, table of contents pages will have lower-case Roman numeral page numbering. If the *Oc* register is set to 1, the .TC macro will not print any page number but will instead reset the *P* register to 1. It is the user's responsibility to give an appropriate page footer to place the page number. Ordinarily the same .PF used in the body of the document will be adequate.

The List of Figures, Tables, etc. pages will be produced separately unless *Cp* is set to 1 which causes these lists to appear on the same page as the table of contents.

10.2 Cover Sheet

.CS [pages] [other] [total] [figs] [tbls] [refs]

The .CS macro generates a cover sheet in either the released paper or Technical Memorandum style (see {6.4} for details of the Memorandum for File cover sheet). All the other information for the cover sheet is obtained from the data given before the .MT macro call {6.9}. If the Technical Memorandum style is used, the .CS macro generates the "Cover Sheet for Technical Memorandum." The data that appear in the lower left corner of the Technical Memorandum cover sheet[9] (the counts of: pages of text, other pages, total pages, figures, tables, and references) are generated automatically (0 is used for the count of "other pages"). These values may be changed by supplying the corresponding arguments to the .CS macro. If the released-paper style is used, all arguments to .CS are ignored.

11. References

There are two macros that delimit the text of references, a string used to automatically number the references, and an optional macro to produce reference pages within the document.

11.1 Automatic Numbering of References

Automatically numbered references may be obtained by typing *(Rf (i.e., invoking the string *Rf*) immediately after the text to be referenced. This places the next sequential reference number (in a smaller point size) enclosed in brackets a half-line above the text to be referenced, as illustrated throughout this document. The reference count is kept in the number register *Rf*.

11.2 Delimiting Reference Text

The .RS and .RF macros are used to delimit the text of each reference.

```
A line of text to be referenced\*(Rf.
.RS [string-name]
reference text
.RF
```

11.3 Subsequent References

.RS takes one argument, a *string-name*. For example:

```
.RS aA
reference text
.RF
```

The string *aA* is assigned the current reference number. It may be used later in the document, as the string call, *(*aA*, to reference text which must be labeled with a prior reference number. The reference is output enclosed in brackets a half-line above the text to be referenced. No .RS/.RF pair is needed for subsequent references.

11.4 Reference Page

A reference page, entitled by default "References", will be generated *automatically* at the end of the document (before the Table of Contents and the Cover Sheet) and will be listed in the Table of Contents. This page contains the reference items (i.e., text enclosed within .RS/.RF pairs). Reference items will be separated by a space (1/2 space) unless the *Ls* register is set to 0 to suppress this spacing. The user may change the reference page title by defining the *Rp* string:

```
.ds Rp "New Title"
```

The .RP (Reference Page) macro may be used to produce reference pages anywhere else within a document (i.e., after each major section); .RP is *not* needed to produce a separate reference page with

default spacings at the end of the document.

.RP [arg1] [arg2]

The two arguments allow the user to control resetting of reference numbering and page skipping.

arg1	*Meaning*
0	reset reference counter (default)
1	do not reset reference counter

arg2	*Meaning*
0	put on separate page (default)
1	do not cause a following .SK
2	do not cause a preceding .SK
3	no .SK before or after

If no .SK is issued by .RP, then a single blank line will separate the References from the following (preceding) text. The user may wish to adjust the spacing. For example, to produce references at the end of each major section:

```
.sp 3
.RP 1 2
.H 1 "Next Section"
```

12. Miscellaneous Features

12.1 Bold, Italic, and Roman Fonts

.B [bold-arg] [previous-font-arg] ...
.I [italic-arg] [previous-font-arg] ...
.R

When called without arguments, .B changes the font to bold and .I changes to underlining (italic). This condition continues until the occurrence of a .R, when the (regular) roman font is restored. Thus:

```
.I
here is some text.
.R
```

yields:

here is some text.

If .B or .I is called with one argument, that argument is printed in the appropriate font (underlined in *nroff* for .I). Then the *previous* font is restored (underlining is turned off in *nroff*). If two or more arguments (maximum 6) are given to a .B or .I, the second argument is then concatenated to the first with no intervening space (1/12 space if the first font is italic), but is printed in the previous font; and the remaining pairs of arguments are similarly alternated. For example:

```
.I italic " text " right -justified
```

produces:

italic text *right*-justified

These macros alternate with the prevailing font at the time they are invoked. To alternate specific pairs of fonts, the following macros are available:

```
.IB   .BI   .IR   .RI   .RB   .BR
```

Each takes a maximum of 6 arguments and alternates the arguments between the specified fonts.

Note that font changes in headings are handled separately {4.2.2.4.1}.

Anyone using a terminal that cannot underline might wish to insert:

```
.rm ul
.rm cu
```

at the beginning of the document to eliminate *all* underlining.

12.2 Justification of Right Margin

.SA [arg]

The .SA macro is used to set right-margin justification for the main body of text. Two justification flags are used: *current* and *default*. -.SA 0 sets both flags to no justification, i.e., it acts like the .na request. .SA 1 is the inverse: it sets both flags to cause justification, just like the .ad request. However, calling .SA *without* an argument causes the *current* flag to be copied from the *default* flag, thus performing either a .na or .ad, depending on what the *default* is. Initially, both flags are set for no justification in *nroff* and for justification in *troff*.

In general, the request .na can be used to ensure that justification is turned off, but .SA should be used to restore justification, rather than the .ad request. In this way, justification or lack thereof for the remainder of the text is specified by inserting .SA 0 or .SA 1 *once* at the beginning of the document.

12.3 SCCS Release Identification

The string *RE* contains the SCCS[10] Release and Level of the current version of MM. For example, typing:

This is version *(RE of the macros.

produces:

This is version 15.130 of the macros.

This information is useful in analyzing suspected bugs in MM. The easiest way to have this number appear in your output is to specify –rD1 {2.4} on the command line, which causes the string *RE* to be output as part of the page header {9.2}.

12.4 Two-Column Output

MM can print two columns on a page:

```
.2C
text and formatting requests (except another .2C)
.1C
```

The .2C macro begins two-column processing which continues until a .1C macro is encountered. In two-column processing, each physical page is thought of as containing two columnar "pages" of equal (but smaller) "page" width. Page headers and footers are *not* affected by two-column processing. The .2C macro does *not* "balance" two-column output.

It is possible to have full-page width footnotes and displays when in two column mode, although the default action is for footnotes and displays to be narrow in two column mode and wide in one column mode. Footnote and display width is controlled by a macro, .WC (Width Control), which takes the following arguments:

arg	*Meaning*
N	Normal default mode (—WF, —FF, —WD, FB)
WF	Wide Footnotes always (even in two column mode)
—WF	DEFAULT: turn off WF (footnotes follow column mode, wide in 1C mode, narrow in 2C mode, unless FF is set)
FF	First Footnote; all footnotes have the same width as the first footnote encountered for that page
—FF	DEFAULT: turn off FF (footnote style follows the settings of WF or —WF)
WD	Wide Displays always (even in two column mode)
—WD	DEFAULT: Displays follow whichever column mode is in effect when the display is encountered
FB	DEFAULT: Floating displays cause a break when output on the current page
—FB	Floating displays on current page do not cause a break

For example: ".WC WD FF" will cause all displays to be wide, and all footnotes on a page to be the same width, while ".WC N" will reinstate the default actions. If conflicting settings are given to .WC the last one is used. That is, ".WC WF —WF" has the effect of ".WC —WF".

12.5 Column Headings for Two-Column Output

NOTE

> This section is intended only for users accustomed to writing formatter macros.

In two-column output, it is sometimes necessary to have headers over each column, as well as headers over the entire page {9}. This is accomplished by redefining the .TP macro {9.12} to provide header lines both for the entire page and for each of the columns. For example:

```
.de TP
.sp 2
.tl 'Page \\\\nP'OVERALL''
.tl ''TITLE''
.sp
.nf
.ta 16C 31R 34 50C 65R
left→center→right→left→center→right    (→ stands for tab character)
→first column→→→second column
.fi
.sp 2
..
```

The above example will produce two lines of page header text plus two lines of headers over each column. The tab stops are for a 65-en overall line length.

12.6 Vertical Spacing

.SP [lines]

There exist several ways of obtaining vertical spacing, all with different effects: the .sp request spaces the number of lines specified, *unless* "no space" (.ns) mode is on, in which case the request is ignored. This mode is set at the end of a page header to eliminate spacing by a .sp or .bp request that happens to occur at the top of a page. This mode can be turned *off* by the .rs ("restore spacing") request.

The .SP macro is used to avoid the accumulation of vertical space by successive macro calls. Several .SP calls in a row produce *not* the sum of their arguments, but their maximum; i.e., the following produces only 3 blank lines:

```
.SP 2
.SP 3
.SP
```

Many MM macros utilize .SP for spacing. For example, ".LE 1" {5.3.2} immediately followed by ".P" {4.1} produces only a single blank line (½ a vertical space) between the end of the list and the following paragraph. An omitted argument defaults to one blank line (*one* vertical space). Negative arguments are not permitted. The argument must be unscaled but fractional amounts are permitted. Like .sp, .SP is also inhibited by the .ns request.

12.7 Skipping Pages

.SK [pages]

The .SK macro skips pages, but retains the usual header and footer processing. If *pages* is omitted, null, or 0, .SK skips to the top of the next page *unless* it is currently at the top of a page, in which case it does nothing. .SK *n* skips *n* pages. That is, .SK always positions the text that follows it at the top of a page, while .SK 1 always leaves one page that is blank except for the header and footer.

12.8 Forcing an Odd Page

.OP

This macro is used to ensure that the text following it begins at the top of an odd-numbered page. If currently at the top of an odd page, no motion takes place. If currently on an even page, text resumes printing at the top of the next page. If currently on an odd page (but not at the top of the page) one blank page is produced, and printing resumes on the page after that.

12.9 Setting the Point Size and Vertical Spacing

In *troff*, the default point size (obtained from the MM register *S* {2.4}) is 10 points, and the vertical spacing is 12 points (i.e., 6 lines per inch). The prevailing point size and vertical spacing may be changed by invoking the .S macro:

.S [point size] [vertical spacing]

The mnemonics D (default value), C (current value), and P (previous value) may be used for both arguments. If an argument is negative, the current value is decremented by the specified amount; if an argument is positive, the current value is incremented by the specified amount; if an argument is unsigned, it is used as the new value; .S without arguments defaults to P. If the first argument is specified but the second is not, then D is used for the vertical spacing; the default value for vertical spacing is always 2 points greater than the *current* point size.[19] A null (" ") value for either argument defaults to C. Thus, if *n* is a numeric value:

```
.S          =  .S P P
.S " " n    =  .S C n
.S n " "    =  .S n C
.S n        =  .S n D
.S " "      =  .S C D
.S " " " "  =  .S C C
.S n n      =  .S n n
```

If the first argument is greater than 99, the default *point size* (10 points) is restored. If the second argument is greater than 99, the default *vertical spacing* (current point size plus 2 points) is used. For example:

```
.S 100      =  .S 10 12
.S 14 111   =  .S 14 16
```

The .SM macro allows one to reduce by 1 point the size of a string:

19. Footnotes {8} are two points *smaller* than the body, with an additional three-point space between footnotes.

.SM string1 [string2] [string3]

If the third argument is omitted, the *first* argument is made smaller and is concatenated with the second argument, if the latter is specified. If all three arguments are present (even if any are null), the *second* argument is made smaller and all three arguments are concatenated. For example:

.SM X	gives	X
.SM X Y	gives	XY
.SM Y X Y	gives	YXY
.SM UNIX	gives	UNIX
.SM UNIX)	gives	UNIX)
.SM (UNIX)	gives	(UNIX)
.SM U NIX " "	gives	UNIX

12.10 Producing Accents

The following strings may be used to produce accents for letters:

	Input	*Output*
Grave accent	e*`	è
Acute accent	e*'	é
Circumflex	o*^	ô
Tilde	n*~	ñ
Cedilla	c*,	ç
Lower-case umlaut	u*:	ü
Upper-case umlaut	U*;	Ü

12.11 Inserting Text Interactively

.RD [prompt] [diversion] [string]

.RD (ReaD insertion) allows a user to stop the standard output of a document and to read text from the standard input until two consecutive new-lines are found. When the new-lines are encountered, normal output is resumed.

.RD follows the formatting conventions in effect. Thus, the examples below assume that the .RD is invoked in no fill mode (.nf).

The first argument is a *prompt* which will be printed at the terminal. If no prompt is given, .RD signals the user with a BEL on terminal output.

The second argument, the name of a *diversion,* allows the user to save all the text typed in after the prompt in a macro whose name is that of the diversion. The third argument, the name of a *string,* allows the user to save for later reference the first line following the prompt in the named string. For example:

```
.RD Name aA bB
```

produces:

```
Name: (user types) J. Jones
16 Elm Rd.,
Piscataway
```

The diverted macro .aA will contain:

```
J. Jones
16 Elm Rd.,
Piscataway
```

The string *bB* (*(*bB*) contains "J. Jones".

A new-line followed by a control-d (EOF) also allows the user to resume normal output.

13. Errors and Debugging

13.1 Error Terminations

When a macro discovers an error, the following actions occur:

- A break occurs.
- To avoid confusion regarding the location of the error, the formatter output buffer (which may contain some text) is printed.
- A short message is printed giving the name of the macro that found the error, the type of error, and the approximate line

number (in the current input file) of the last processed input line. (All the error messages are explained in Appendix D.)

- Processing terminates, unless the register *D* {2.4} has a positive value. In the latter case, processing continues even though the output is guaranteed to be deranged from that point on.

NOTE

The error message is printed by writing it directly to the user's terminal. If an output filter, such as *300(1), 450(1),* or *hp(1)* is being used to post-process *nroff* output, the message may be garbled by being intermixed with text held in that filter's output buffer.

If any of *cw(1), eqn(1)/neqn,* and *tbl(1)* are being used, and if the *–olist option of the formatter causes the last page of the document not* to be printed, a harmless "broken pipe" message may result.

13.2 Disappearance of Output

This usually occurs because of an unclosed diversion (e.g., missing .DE or .FE). Fortunately, the macros that use diversions are careful about it, and they check to make sure that illegal nestings do not occur. If any message is issued about a missing .DE or .FE, the appropriate action is to search backwards from the termination point looking for the corresponding .DF, .DS, or .FS.

The following command:

```
grep -n "^\.[EDFRT][EFNQS]" files ...
```

prints all the .DF, .DS, .DE, .EQ, .EN, .FS, .FE, .RS, .RF, .TS, and .TE macros found in *files ...*, each preceded by its file name and the line number in that file. This listing can be used to check for illegal nesting and/or omission of these macros.

14. Extending and Modifying the Macros

14.1 Naming Conventions

In this section, the following conventions are used to describe names:

n: digit
a: lower-case letter
A: upper-case letter
x: n, a, or A: i.e., any letter or digit (any alphanumeric character)
s: special character (any non-alphanumeric character)

All other characters are literals (i.e., stand for themselves).

Note that *request, macro,* and *string* names are kept by the formatters in a single internal table, so that there must be no duplication among such names. *Number register* names are kept in a separate table.

14.1.1 Names Used by Formatters

requests:	aa (most common) an (only one, currently: c2)
registers:	aa (normal) .x (normal) .s (only one, currently: .$) a. (only one, currently: c.) % (page number)

14.1.2 Names Used by MM

macros and strings:	A, AA, Aa (accessible to users; e.g., macros P and HU, strings F, BU, and Lt) nA (accessible to users; only two, currently: 1C and 2C) aA (accessible to users; only one, currently: nP) s (accessible to users; only the seven accents, currently {12.10}))x, }x,]x, >x, ?x (internal)

registers: An, Aa (accessible to users; e.g., H1, Fg)
A (accessible to users; meant to be set on the command line; e.g., C)
:x, ;x, #x, ?x, !x (internal)

14.1.3 Names Used by CW, EQN/NEQN, and TBL

Cw(1), the constant-width font preprocessor for *troff*, uses the following five macro names: .CD, .CN, .CP, .CW, and .PC; it also uses the number register names *cE* and *cW*. The equation preprocessors, *eqn*(1) and *neqn* use registers *and* string names of the form *nn*. The table preprocessor, *tbl*(1), uses T&, T#, and TW, and names of the form:

a– a+ a| nn na ^a #a #s

14.1.4 User-Definable Names.

Given the above, what is left for user extensions? To avoid "collisions" with already used names, use names that consist either of a single lower-case letter, or of a lower-case letter followed by a character *other than* a lower-case letter (remembering, however, that the names .c2 and .nP are already used). The following is a possible user naming convention:

macros: aA (e.g., bG, kW)
strings: as (e.g., c), f], p})
registers: a (e.g., f, t)

14.2 Sample Extensions

14.2.1 Appendix Headings

The following is a way of generating and numbering appendix headings:

```
.nr Hu 1
.nr a 0
.de aH
.nr a +1
.nr P 0
.PH "'''Appendix \\na-\\\\nP'"
.SK
.HU "\\$1"
..
```

After the above initialization and definition, each call of the form ".aH "title"" begins a new page (with the page header changed to "Appendix *a-n*") and generates an unnumbered heading of *title*, which, if desired, can be saved for the table of contents. Those who wish apppendix titles to be centered must, in addition, set the register *Hc* to 1 {4.2.2.3}.

14.2.2 Hanging Indent with Tabs

The following example illustrates the use of the hanging-indent feature of variable-item lists {5.3.3.6}. First, a user-defined macro is built to accept four arguments that make up the *mark*. In the output, each argument is to be separated from the previous one by a tab; tab settings are defined later. Since the first argument may begin with a period or apostrophe, the "\&" is used so that the formatter will not interpret such a line as a formatter request or macro call.[20] The "\t" is translated by the formatter into a tab. The "\c" is used to concatenate the input *text* that follows the macro call to the line built by the macro. The macro and an example of its use are:

20. The two-character sequence "\&" is understood by the formatters to be a "zero-width" space, i.e., it causes no output characters to appear, but it removes the special meaning of a leading period or apostrophe.

```
.de aX
.LI
\&\\$1\t\\$2\t\\$3\t\\$4\t\c
..
  .
  .
  .
.ta 7n 15n 25n 30n
.VL 30
.aX .nh off \- no
No hyphenation.
Automatic hyphenation is turned off.
Words containing hyphens
(e.g., mother-in-law) may still be split across lines.
.aX .hy on \- no
Hyphenate.
Automatic hyphenation is turned on.
.aX .hc\□c none none no                    (□ stands for a space)
Hyphenation indicator character is set to "c" or removed.
During text processing the indicator is suppressed
and will not appear in the output.
Prepending the indicator to a word has the effect
of preventing hyphenation of that word.
.LE
```

The resulting output is:

.nh	off	—	no	No hyphenation. Automatic hyphenation is turned off. Words containing hyphens (e.g., mother-in-law) may still be split across lines.
.hy	on	—	no	Hyphenate. Automatic hyphenation is turned on.
.hc c	none	none	no	Hyphenation indicator character is set to "c" or removed. During text processing the indicator is suppressed and will not appear in the output. Prepending the indicator to a word has the effect of preventing hyphenation of that word.

15. Conclusion

The following are the qualities that we have tried to emphasize in MM, in approximate order of importance:

- *Robustness in the face of error*—A user need not be an *nroff/troff* expert to use these macros. When the input is incorrect, either the macros attempt to make a reasonable interpretation of the error, or a message describing the error is produced. We have tried to minimize the possibility that a user would get cryptic system messages or strange output as a result of simple errors.
- *Ease of use for simple documents*—It is not necessary to write complex sequences of commands to produce simple documents. Reasonable default values are provided, where at all possible.
- *Parameterization*—There are many different preferences in the area of document styling. Many parameters are provided so that users can adapt the output to their respective needs over a wide range of styles.
- *Extension by moderately expert users*—We have made a strong effort to use mnemonic naming conventions and consistent techniques in the construction of the macros. Naming conventions are given so that a user can add new macros or redefine existing ones, if necessary.
- *Device independence*—The most common use of MM is to print documents on hard-copy typewriter terminals, using the *nroff* formatter. The macros can be used conveniently with both 10- and 12-pitch terminals. In addition, output can be scanned with an appropriate CRT terminal. The macros have been constructed to allow compatibility with *troff*, so that output can be produced both on typewriter-like terminals and on a phototypesetter.
- *Minimization of input*—The design of the macros attempts to minimize repetitive typing. For example, if a user wants to have a blank line after all first- or second-level headings, he or she need only set a specific parameter *once* at the beginning of a document, rather than add a blank line after each such heading.
- *Decoupling of input format from output style*—There is but one way to prepare the input text, although the user may obtain a number of output styles by setting a few global flags. For

example, the .H macro is used for all numbered headings, yet the actual output style of these headings may be made to vary from document to document or, for that matter, within a single document.

Acknowledgements

We are indebted to T. A. Dolotta for his continuing guidance during the development of MM. We also thank our many users who have provided much valuable feedback, both about the macros and about this manual. Many of the features of MM are patterned after similar features in a number of earlier macro packages, and, in particular, after one implemented by M. E. Lesk. Finally, because MM often approaches the limits of what is possible with the text formatters, during the implementation of MM we have generated atypical requirements and encountered unusual problems; we thank the late J. F. Ossanna for his willingness to add new features to the formatters and to invent ways of having the formatters perform unusual but desired actions.

REFERENCES

1. Dolotta, T. A., Olsson, S. B., and Petruccelli, A. G. (eds.). *UNIX User's Manual*–Release 3.0, Bell Laboratories (June 1980).
2. Ossanna, J. F. *NROFF/TROFF User's Manual.* Bell Laboratories, October 1976.
3. Kernighan, B. W. *UNIX for Beginners.* Bell Laboratories, October 1974.
4. Kernighan, B. W. *A Tutorial Introduction to the UNIX Text Editor.* Bell Laboratories, October 1974.
5. Kernighan, B. W. *A TROFF Tutorial.* Bell Laboratories, August 1976.
6. Kernighan, B. W. and Cherry, L. L. *Typesetting Mathematics—User's Guide (Second Edition).* Bell Laboratories, August 1978.
7. Lesk, M. *TBL—A Program to Format Tables.* Bell Laboratories, September 1977.

8. Smith, D. W. and Piskorik, E. M. *Typing Documents with MM.* Bell Laboratories, April 1980.

9. Bell Laboratories Methods and Systems Department. *Office Guide.* Unpublished Memorandum, Bell Laboratories, April 1972 (as revised).

10. Bonanni, L. E. and Salemi, C. A. *The Source Code Control System User's Guide.* Bell Laboratories, April 1979.

Appendix A: User-defined List Structures

NOTE

This appendix is intended only for users accustomed to writing formatter macros.

If a large document requires complex list structures, it is useful to be able to define the appearance for each list level only once, instead of having to define it at the beginning of each list. This permits consistency of style in a large document. For example, a generalized list-initialization macro might be defined in such a way that what it does depends on the list-nesting level in effect at the time the macro is called. Suppose that levels 1 through 5 of lists are to have the following appearance:

A.

[1]

•

a)

+

The following code defines a macro (.aL) that always begins a new list and determines the type of list according to the current list level. To understand it, you should know that the number register *:g* is used by the MM list macros to determine the current list level; it is 0 if there is no currently active list. Each call to a list-initialization macro increments *:g*, and each .LE call decrements it.

```
.de aL
'\"       register g is used as a local temporary to save :g
          before it is changed below
.nr g \\n(:g
.if \\ng=0 .AL A \" give me an A.
.if \\ng=1 .LB \\n(Li 0 1 4 \" give me a [1]
.if \\ng=2 .BL \" give me a bullet
.if \\ng=3 .LB \\n(Li 0 2 2 a \" give me an a)
.if \\ng=4 .ML + \" give me a +
..
```

This macro can be used (in conjunction with .LI and .LE) instead of .AL, .RL, .BL, .LB, and .ML. For example, the following input:

```
.aL
.LI
first line.
.aL
.LI
second line.
.LE
.LI
third line.
.LE
```

will yield:

A. first line.

[1] second line.

B. third line.

There is another approach to lists that is similar to the .H mechanism. The list-initialization, as well as the .LI and the .LE macros are all included in a single macro. That macro (called .bL below) requires an argument to tell it what level of item is required; it adjusts the list level by either beginning a new list or setting the list level back to a previous value, and then issues a .LI macro call to produce the item:

```
.de bL
.ie \\n(.$ .nr g \\$1 \" if there is an argument, that is the level
.el .nr g \\n(:g \" if no argument, use current level
.if \\ng-\\n(:g>1 .)D "**ILLEGAL SKIPPING OF LEVEL"
\" increasing level by more than 1
.if \\ng>\\n(:g \{.aL \\ng-1 \" if g > :g, begin new list
. nr g \\n(:g\} \" and reset g to current level (.aL changes g)
.if \\n(:g>\\ng .LC \\ng \" if :g > g, prune back to correct level
'\"  if :g = g, stay within current list
.LI \" in all cases, get out an item
..
```

For .bL to work, the previous definition of the .aL macro must be changed to obtain the value of *g* from its argument, rather than from *:g*. Invoking .bL without arguments causes it to stay at the current list level. The MM .LC macro (List Clear) removes list descriptions until the level is less than or equal to that of its argument. For example, the .H macro includes the call ".LC 0". If text is to be resumed at the end of a list, insert the call ".LC 0" to clear out the lists completely. The example below illustrates the relatively small amount of input needed by this approach. The input text:

```
The quick brown fox jumped over the lazy dog's back.
.bL 1
first line.
.bL 2
second line.
.bL 1
third line.
.bL
fourth line.
.LC 0
fifth line.
```

yields:

The quick brown fox jumped over the lazy dog's back.

A. first line.

   [1] second line.

B. third line.

C. fourth line.

fifth line.

Appendix B: Sample Footnotes

The following example illustrates several footnote styles and both labeled and automatically-numbered footnotes. The actual input for the immediately following text and for the footnotes at the bottom of this page is shown on the following page:

With the footnote style set to the *nroff* default, we process a footnote[1] followed by another one.***** Using the .FD macro, we changed the footnote style to hyphenate, right margin justification, indent, and left justify the label. Here is a footnote,[2] and another.† The footnote style is now set, again via the .FD macro, to no hyphenation, no right margin justification, no indentation, and with the label left-justified. Here comes the final one.[3]

1. This is the first footnote text example (.FD 10). This is the default style for *nroff*. The right margin is *not* justified. Hyphenation is *not* permitted. The text is indented, and the automatically generated label is *right*-justified in the text-indent space.

***** This is the second footnote text example (.FD 10). This is also the default *nroff* style but with a long footnote label provided by the user.

2. This is the third footnote example (.FD 1). The right margin is justified, the footnote text is indented, the label is *left*-justified in the text-indent space. Although not necessarily illustrated by this example, hyphenation is permitted. The quick brown fox jumped over the lazy dog's back.

† This is the fourth footnote example (.FD 1). The style is the same as the third footnote.

3. This is the fifth footnote example (.FD 6). The right margin is *not* justified, hyphenation is *not* permitted, the footnote text is *not* indented, and the label is placed at the beginning of the first line. The quick brown fox jumped over the lazy dog's back. Now is the time for all good men to come to the aid of their country.

```
.FD 10
With the footnote style set to the
.I nroff
default, we process a footnote\*F
.FS
This is the first footnote text example (.FD 10).  This is the default style for
.I nroff.
The right margin is
.I not
justified.  Hyphenation is
.I not
permitted.  The text is indented, and the automatically generated label is
.I right -justified
in the text-indent space.
.FE
followed by another one.*****\□                    (□ stands for a space)
.FS *****
This is the second footnote text example (.FD 10).
This is also the default
.I nroff
style but with a long footnote label provided by the user.
.FE
.FD 1
Using the .FD macro, we changed the footnote style to
hyphenate, right margin justification, indent, and left justify the label.
Here is a footnote,\*F
.FS
This is the third footnote example (.FD 1).
The right margin is justified, the footnote text is indented, the label is
.I left -justified
in the text-indent space.
Although not necessarily illustrated by this example, hyphenation is permitted.
The quick brown fox jumped over the lazy dog's back.
.FE
and another.\(dg\□
.FS \(dg
This is the fourth footnote example (.FD 1).
The style is the same as the third footnote.
.FE
.FD 6
The footnote style is now set, again via the .FD macro, to no hyphenation, no right margin
justification, no indentation, and with the label left-justified.
Here comes the final one.\*F\□
.FS
This is the fifth footnote example (.FD 6).  The right margin is
.I not
justified, hyphenation is
.I not
permitted, the footnote text is
.I not
indented, and the label is placed at the beginning of the first line.
The quick brown fox jumped over the lazy dog's back.
Now is the time for all good men to come to the aid of their country.
.FE
```

Appendix C: Sample Letter

Input:

```
.ND "May 31, 1979"
.TL 334455
Out-of-Hours Course Description
.AU "D. W. Stevenson" DWS PY 9876 5432 1X-123
.MT 0
.DS
J. M. Jones:
.DE
.P
Please use the following description for the Out-of-Hours course
.I
Document Preparation on the UNIX*
.R
.FS *
UNIX is a trademark of Bell Laboratories.
.FE
.I "Time-Sharing System:"
.P
The course is intended for clerks, typists, and others who intend to use the
UNIX system for preparing documentation.
The course will cover such topics as:
.VL 18
.LI Environment:
utilizing a time-sharing computer system;
accessing the system; using appropriate output terminals.
.LI Files:
how text is stored on the system; directories; manipulating files.
.LI "Text editing:"
how to enter text so that subsequent revisions are easier to make;
how to use the editing system to add, delete, and move lines
of text; how to make corrections.
.LI "Text processing:"
basic concepts; use of general-purpose formatting packages.
.LI "Other facilities:"
additional capabilities useful to the typist such as the
.I "spell, diff,"
and
.I grep
commands, and a desk-calculator package.
.LE
.SG jrm
.NS
S. P. Lename
H. O. Del
M. Hill
.NE
```

Bell Laboratories

subject: Out-of-Hours Course Description
Case: 334455

date: May 31, 1979

from: D. W. Stevenson
PY 9876
1X-123 x5432

J. M. Jones:

Please use the following description for the Out-of-Hours course ``Document Preparation on the UNIX* Time-Sharing System":

The course is intended for clerks, typists, and others who intend to use the UNIX system for preparing documentation. The course will cover such topics as:

Environment:	utilizing a time-sharing computer system; accessing the system; using appropriate output terminals.
Files:	how text is stored on the system; directories; manipulating files.
Text editing:	how to enter text so that subsequent revisions are easier to make; how to use the editing system to add, delete, and move lines of text; how to make corrections.
Text processing:	basic concepts; use of general-purpose formatting packages.
Other facilities:	additional capabilities useful to the typist such as the spell, diff, and grep commands, and a desk-calculator package.

PY-9876-DWS-jrm

D. W. Stevenson

Copy to
S. P. Lename
H. O. Del
M. Hill

* UNIX is a trademark of Bell Laboratories.

Bell Laboratories

subject: **Out-of-Hours Course Description - Case 334455**

date: **May 31, 1979**

from: **D. W. Stevenson**
PY 9876
1X-123 x5432

J. M. Jones:

Please use the following description for the Out-of-Hours course *Document Preparation on the UNIX* Time-Sharing System:*

The course is intended for clerks, typists, and others who intend to use the UNIX system for preparing documentation. The course will cover such topics as:

Environment:	utilizing a time-sharing computer system; accessing the system; using appropriate output terminals.
Files:	how text is stored on the system; directories; manipulating files.
Text editing:	how to enter text so that subsequent revisions are easier to make; how to use the editing system to add, delete, and move lines of text; how to make corrections.
Text processing:	basic concepts; use of general-purpose formatting packages.
Other facilities:	additional capabilities useful to the typist such as the *spell*, *diff*, and *grep* commands, and a desk-calculator package.

PY-9876-DWS-jrm **D. W. Stevenson**

Copy to
S. P. Lename
H. O. Del
M. Hill

---

* UNIX is a trademark of Bell Laboratories.

Appendix D: Error Messages

I. MM Error Messages

An MM error message has a standard part followed by a variable part. The standard part has the form:

ERROR:(*filename*)input line *n*:

The variable part consists of a descriptive message, usually beginning with a macro name. The variable parts are listed below in alphabetical order by macro name, each with a more complete explanation:[21]

Check TL, AU, AS, AE, MT sequence	The correct order of macros at the start of a memorandum is shown in {6.9}. Something has disturbed this order. If .AS 2 was used, then the error message will be "Check TL, AU, AS, AE, NS, NE, MT sequence".
CS:cover sheet too long	The text of the cover sheet is too long to fit on one page. The abstract should be reduced or the indent of the abstract should be decreased {6.4}.
DE:no DS or DF active	.DE has been encountered but there has not been a previous .DS or .DF to match it.
DF:illegal inside TL or AS	Displays are not allowed in the title or abstract.

21. This list is set up by ".LB 37 0 2 0" {5.4}.

DF:missing DE	.DF occurs within a display, i.e., a .DE has been omitted or mistyped.
DF:missing FE	A display starts inside a footnote. The likely cause is the omission (or misspelling) of a .FE to end a previous footnote.
DF:too many displays	More than 26 floating displays are active at once, i.e., have been accumulated but not yet output.
DS:illegal inside TL or AS	Displays are not allowed in the title or abstract.
DS:missing DE	.DS occurs within a display, i.e., a .DE has been omitted or mistyped.
DS:missing FE	A display starts inside a footnote. The likely cause is the omission (or misspelling) of a .FE to end a previous footnote.
FE:no FS active	.FE has been encountered with no previous .FS to match it.
FS:missing DE	A footnote starts inside a display, i.e., a .DS or .DF occurs without a matching .DE.
FS:missing FE	A previous .FS was not matched by a closing .FE, i.e., an attempt is being made to begin a footnote inside another one.
H:bad arg:value	The first argument to .H must be a single digit from 1 to 7, but *value* has been supplied instead.

H:missing arg	.H needs at least 1 argument.
H:missing DE	A heading macro (.H or .HU) occurs inside a display.
H:missing FE	A heading macro (.H or .HU) occurs inside a footnote.
HU:missing arg	.HU needs 1 argument.
LB:missing arg(s)	.LB requires at least 4 arguments.
LB:too many nested lists	Another list was started when there were already 6 active lists.
LE:mismatched	.LE has occurred without a previous .LB or other list-initialization macro {5.3.3}. Although this is not a fatal error, the message is issued because there almost certainly exists some problem in the preceding text.
LI:no lists active	.LI occurs without a preceding list-initialization macro. The latter has probably been omitted, or has been separated from the .LI by an intervening .H or .HU.
ML:missing arg	.ML requires at least 1 argument.
ND:missing arg	.ND requires 1 argument.
RF:no RS active	.RF has been encountered with no previous .RS to match it.
RP:missing RF	A previous .RS was not matched by a closing .RF.

RS:missing RF	A previous .RS was not matched by a closing .RF.
S:bad arg:value	The incorrect argument *value* has been given for .S, see {12.9}.
SA:bad arg:value	The argument to .SA (if any) must be either 0 or 1. The incorrect argument is shown as *value*.
SG:missing DE	.SG occurs inside a display.
SG:missing FE	.SG occurs inside a footnote.
SG:no authors	.SG occurs without any previous .AU macro(s).
VL:missing arg	.VL requires at least 1 argument.
WC:unknown option	An incorrect argument has been given to .WC, see {12.4}.

II. Formatter Error Messages

Most messages issued by the formatter are self-explanatory. Those error messages over which the *user* has (some) control are listed below. Any other error messages should be reported to the local system-support group.

"Cannot do ev" is caused by *(a)* setting a page width that is negative or extremely short; *(b)* setting a page length that is negative or extremely short; *(c)* reprocessing a macro package (e.g., performing a .so formatter request on a macro package that was already requested on the command line); and *(d)* requesting the *troff* —s1 option on a document that is longer than ten pages.

"Cannot execute *filename* " is given by the .! request if it cannot find the *filename*.

"Cannot open *filename* " is issued if one of the files in the list of files to be processed cannot be opened.

"Exception word list full" indicates that too many words have been specified in the hyphenation exception list (via .hw requests).

"Line overflow" means that the output line being generated was too long for the formatter's line buffer. The excess was discarded. See the "Word overflow" message below.

"Non-existent font type" means that a request has been made to mount an unknown font.

"Non-existent macro file" means that the requested macro package does not exist.

"Non-existent terminal type" means that the terminal options refers to an unknown terminal type.

"Out of temp file space" means that additional temporary space for macro definitions, diversions, etc. cannot be allocated. This message often occurs because of unclosed diversions (missing .FE or .DE), unclosed macro definitions (e.g., missing ".."), or a huge table of contents.

"Too many page numbers" is issued when the list of pages specified to the formatter *–o* option is too long.

"Too many number registers" means that the pool of number register names is full. Unneeded registers can be deleted by using the .rr request.

"Too many string/macro names" is issued when the pool of string and macro names is full. Unneeded strings and macros can be deleted using the .rm request.

"Word overflow" means that a word being generated exceeded the formatter's word buffer. The excess characters were discarded. A likely cause for this and for the "Line overflow" message above are very long lines or words generated through the misuse of \c or of the .cu request, or very long equations produced by *eqn*(1)/*neqn*.

Appendix E: Summary of Macros, Strings, and Number Registers

I. Macros

The following is an alphabetical list of macro names used by MM. The first line of each item gives the name of the macro, a brief description, and a reference to the section in which the macro is described. The second line gives a prototype call of the macro.

Macros marked with an asterisk are *not*, in general, invoked directly by the user. Rather, they are "user exits" *defined* by the user and called by the MM macros from inside header, footer, or other macros.

1C — One-column processing {12.4}
.1C

2C — Two-column processing {12.4}
.2C

AE — Abstract end {6.4}
.AE

AF — Alternate format of "Subject/Date/From" block {6.7.2}
.AF [company-name]

AL — Automatically-incremented list start {5.3.3.1}
.AL [type] [text-indent] [1]

AS — Abstract start {6.4}
.AS [arg] [indent]

AT — Author's title {6.2}
.AT [title] ...

AU — Author information {6.2}
.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]

AV — Approval signature {6.11.3}
.AV [name]

B — Bold {12.1}
.B [bold-arg] [previous-font-arg] [bold] [prev] [bold] [prev]

BE — Bottom Block End {9.13}
.BE

BI — Bold/Italic {12.1}
.BI [bold-arg] [italic-arg] [bold] [italic] [bold] [italic]

BL Bullet list start {5.3.3.2}
.BL [text-indent] [1]

BR Bold/Roman {12.1}
.BR [bold-arg] [Roman-arg] [bold] [Roman] [bold] [Roman]

BS Bottom Block Start {9.13}
.BS

CS Cover sheet {10.2}
.CS [pages] [other] [total] [figs] [tbls] [refs]

DE Display end {7.1}
.DE

DF Display floating start {7.2}
.DF [format] [fill] [right-indent]

DL Dash list start {5.3.3.3}
.DL [text-indent] [1]

DS Display static start {7.1}
.DS [format] [fill] [right-indent]

EC Equation caption {7.5}
.EC [title] [override] [flag]

EF Even-page footer {9.6}
.EF [arg]

EH Even-page header {9.3}
.EH [arg]

EN End equation display {7.4}
.EN

EQ Equation display start {7.4}
.EQ [label]

EX Exhibit caption {7.5}
.EX [title] [override] [flag]

FC Formal closing {6.11}
.FC [closing]

FD Footnote default format {8.3}
.FD [arg] [1]

FE Footnote end {8.2}
.FE

FG Figure title {7.5}
.FG [title] [override] [flag]

FS Footnote start {8.2}
.FS [label]

H Heading—numbered {4.2}
.H level [heading-text] [heading-suffix]

HC Hyphenation character {3.4}
.HC [hyphenation-indicator]

HM Heading mark style (Arabic or Roman numerals, or letters) {4.2.2.5}
.HM [arg1] ... [arg7]

HU Heading—unnumbered {4.3}
.HU heading-text

HX * Heading user exit X (before printing heading) {4.6}
.HX dlevel rlevel heading-text

HY * Heading user exit Y (before printing heading) {4.6}
.HY dlevel rlevel heading-text

HZ * Heading user exit Z (after printing heading) {4.6}
.HZ dlevel rlevel heading-text

I Italic (underline in *nroff*) {12.1}
.I [italic-arg] [previous-font-arg] [italic] [prev] [italic] [prev]

IB Italic/Bold {12.1}
.IB [italic-arg] [bold-arg] [italic] [bold] [italic] [bold]

IR Italic/Roman {12.1}
.IR [italic-arg] [Roman-arg] [italic] [Roman] [italic] [Roman]

LB List begin {5.4}
.LB text-indent mark-indent pad type [mark] [LI-space] [LB-space]

LC List-status clear {Appendix A}
.LC [list-level]

LE List end {5.3.2}
.LE [1]

LI List item {5.3.1}
.LI [mark] [1]

ML Marked list start {5.3.3.4}
.ML mark [text-indent] [1]

MT Memorandum type {6.6}
.MT [type] [addressee] *or* .MT [4] [1]

ND New date {6.7.1}
.ND new-date

NE Notation end {6.11.2}
.NE

NS Notation start {6.11.2}
.NS [arg]

nP Double-line indented paragraphs {4.1}
.nP

OF Odd-page footer {9.7}
.OF [arg]

OH Odd-page header {9.4}
.OH [arg]

OK Other keywords for the Technical Memorandum cover sheet {6.5}
.OK [keyword] ...

OP Odd page {12.8}
.OP

P Paragraph {4.1}
.P [type]

PF Page footer {9.5}
.PF [arg]

PH Page header {9.2}
.PH [arg]

PM Proprietary Marking {9.15}
.PM [code]

PX * Page-header user exit {9.12}
.PX

R Return to regular (roman) font (end underlining in *nroff*) {12.1}
.R

RB Roman/Bold {12.1}
.RB [Roman-arg] [bold-arg] [Roman] [bold] [Roman]

[bold]

RD — Read insertion from terminal {12.11}
.RD [prompt] [diversion] [string]

RF — Reference end {11.2}
.RF

RI — Roman/Italic {12.1}
.RI [Roman-arg] [italic-arg] [Roman] [italic] [Roman] [italic]

RL — Reference list start {5.3.3.5}
.RL [text-indent] [1]

RP — Produce Reference Page {11.4}
.RP [arg] [arg]

RS — Reference start {11.2}
.RS [string-name]

S — Set *troff* point size and vertical spacing {12.9}
.S [size] [spacing]

SA — Set adjustment (right-margin justification) default {12.2}
.SA [arg]

SG — Signature line {6.11.1}
.SG [arg] [1]

SK — Skip pages {12.7}
.SK [pages]

SM — Make a string smaller {12.9}
.SM string1 [string2] [string3]

SP — Space vertically {12.6}
.SP [lines]

TB — Table title {7.5}
.TB [title] [override] [flag]

TC — Table of contents {10.1}
.TC [slevel] [spacing] [tlevel] [tab] [head1] [head2] [head3] [head4] [head5]

TE — Table end {7.3}
.TE

TH — Table header {7.3}
.TH [N]

TL Title of memorandum {6.1}
.TL [charging-case] [filing-case]

TM Technical Memorandum number(s) {6.3}
.TM [number] ...

TP * Top-of-page macro {9.12}
.TP

TS Table start {7.3}
.TS [H]

TX * Table-of-contents user exit {10.1}
.TX

TY * Table-of-contents user exit (suppresses "CONTENTS") {10.1}
.TY

VL Variable-item list start {5.3.3.6}
.VL text-indent [mark-indent] [1]

VM Vertical margins {9.14}
.VM [top] [bottom]

WC Width Control {12.4}
.WC [format]

II. Strings

The following is an alphabetical list of string names used by MM, giving for each a brief description, section reference, and initial (default) value(s). See {1.4} for notes on setting and referencing strings.

BU Bullet {3.7}
nroff: ⊕
troff: •

Ci Table-of-contents indent list, up to seven args for heading levels (must be scaled) {10.1}

DT Date (current date, unless overridden) {6.7.1}
Month day, year (e.g.,)

EM Em dash string, produces an em dash in *troff* and a double hyphen in *nroff* {3.8}.

F Footnote numberer {8.1}
nroff: \u\\n+(:p\d

troff: \v'-.4m'\s-3\\n+(:p\s0\v'.4m'

HF Heading font list, up to seven codes for heading levels 1 through 7 {4.2.2.4.1}
3 3 2 2 2 2 2 (levels 1 and 2 bold, 3-7 underlined in *nroff*, and italic in *troff*)

HP Heading point size list, up to seven codes for heading levels 1 through 7 {4.2.2.4.3}

Le Title for LIST OF EQUATIONS {7.6}

Lf Title for LIST OF FIGURES {7.6}

Lt Title for LIST OF TABLES {7.6}

Lx Title for LIST OF EXHIBITS {7.6}

RE SCCS Release and Level of MM {12.3}
Release.Level (e.g., 15.130)

Rf Reference numberer {11.1}

Rp Title for References {11.4}

Tm Trademark string; places the letters "TM" one half-line above the text that it follows {3.9}.

Seven accent strings are also available {12.10}.

If the released-paper style is used, then, in addition to the above strings, certain BTL location codes are defined as strings; these location strings are needed only until the .MT macro is called {6.8}. Currently, the following are recognized: AK, AL, ALF, CB, CH, CP, DR, FJ, HL, HO, HOH, HP, IH, IN, INH, IW, MH, MV, PY, RD, RR, WB, WH, and WV.

III. Number Registers

This section provides an alphabetical list of register names, giving for each a brief description, section reference, initial (default) value, and the legal range of values (where [m:n] means values from m to n inclusive).

Any register having a single-character name can be set from the command line. An asterisk attached to a register name indicates that that register can be set *only* from the command line or *before* the MM macro definitions are read by the formatter {2.4, 2.5}. See {1.4} for notes on setting and referencing registers.

A * Handles preprinted forms and the Bell System logo {2.4}
0, [0:2]

Au Inhibits printing of author's location, department, room, and extension in the "from" portion of a memorandum {6.2}
1, [0:1]

C * Copy type (Original, DRAFT, etc.) {2.4}
0 (Original), [0:4]

Cl Contents level (i.e., level of headings saved for table of contents) {4.4}
2, [0:7]

Cp Placement of List of Figures, etc. {10.1}
1 (on separate pages), [0:1]

D * Debug flag {2.4}
0, [0:1]

De Display eject register for floating dislays {7.2}
0, [0:1]

Df Display format register for floating displays {7.2}
5, [0:5]

Ds Static display pre- and post-space {7.1}
1, [0:1]

E * Controls font of the Subject/Date/From fields {2.4}
0 (*nroff*) 1 (*troff*), [0:1]

Ec Equation counter, used by .EC macro {7.5}
0, [0:?], incremented by 1 for each .EC call.

Ej Page-ejection flag for headings {4.2.2.1}
0 (no eject), [0:7]

Eq Equation label placement {7.4}
0 (right-adjusted), [0:1]

Ex Exhibit counter, used by .EX macro {7.5}
0, [0:?], incremented by 1 for each .EX call.

Fg Figure counter, used by .FG macro {7.5}
0, [0:?], incremented by 1 for each .FG call.

Fs Footnote space (i.e., spacing between footnotes) {8.4}
1, [0:?]

H1-H7 Heading counters for levels 1-7 {4.2.2.5}
0, [0:?], incremented by .H of corresponding level or .HU if at level given by register *Hu*. H2-H7 are reset to 0 by any heading at a lower-numbered level.

Hb Heading break level (after .H and .HU) {4.2.2.2}
2, [0:7]

Hc Heading centering level for .H and .HU {4.2.2.3}
0 (no centered headings), [0:7]

Hi Heading temporary indent (after .H and .HU) {4.2.2.2}
1 (indent as paragraph), [0:2]

Hs Heading space level (after .H and .HU) {4.2.2.2}
2 (space only after .H 1 and .H 2), [0:7]

Ht Heading type (for .H: single or concatenated numbers) {4.2.2.5}
0 (concatenated numbers: 1.1.1, etc.), [0:1]

Hu Heading level for unnumbered heading (.HU) {4.3}
2 (.HU at the same level as .H 2), [0:7]

Hy Hyphenation control for body of document {3.4}
0 (automatic hyphenation off), [0:1]

L * Length of page {2.4}
66, [20:?] (11i, [2i:?] in *troff*)[22]

Le List of Equations {7.6}
0 (list not produced) [0:1]

Lf List of Figures {7.6}
1 (list produced) [0:1]

Li List indent {5.3.3.1}
6 (*nroff*) 5 (*troff*), [0:?]

Ls List spacing between items by level {5.3.3.1}
6 (spacing between all levels) [0:6]

Lt List of Tables {7.6}
1 (list produced) [0:1]

22. For *nroff*, these values are *unscaled* numbers representing lines or character positions; for *troff*, these values must be *scaled*.

Lx — List of Exhibits {7.6}
1 (list produced) [0:1]

N * — Numbering style {2.4}
0, [0:5]

Np — Numbering style for paragraphs {4.1}
0 (unnumbered) [0:1]

O * — Offset of page {2.4}
.75i, [0:?] (0.5i, [0i:?] in *troff*)[22]

Oc — Table of Contents page numbering style {10.1}
0 (lower-case Roman), [0:1]

Of — Figure caption style {7.5}
0 (period separator), [0:1]

P — Page number, managed by MM {2.4}
0, [0:?]

Pi — Paragraph indent {4.1}
5 (*nroff*) 3 (*troff*), [0:?]

Ps — Paragraph spacing {4.1}
1 (one blank space between paragraphs), [0:?]

Pt — Paragraph type {4.1}
0 (paragraphs always left-justified), [0:2]

Pv — "PRIVATE" header {9.16}
0 (not printed), [0:2]

Rf — Reference counter, used by .RS macro {11.1}
0, [0:?], incremented by 1 for each .RS call.

S * — *Troff* default point size {2.4}
10, [6:36]

Si — Standard indent for displays {7.1}
5 (*nroff*) 3 (*troff*), [0:?]

T * — Type of *nroff* output device {2.4}
0, [0:2]

Tb — Table counter, used by .TB macro {7.5}
0, [0:?], incremented by 1 for each .TB call.

U * — Underlining style (*nroff*) for .H and .HU {2.4}
0 (continuous underline when possible), [0:1]

W * — Width of page (line and title length) {2.4}
6i, [10:1365] (6i, [2i:7.54i] in *troff*)[22]

10

Using the –ms Macros with Troff and Nroff

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.

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.

Source: M. E. Lesk, *Typing Documents on the UNIX System: Using the –ms Macros with Troff and Nroff* (Murray Hill, N.J.: Bell Laboratories, 1978).

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.

CAUTION

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.

CAUTION

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

* Like this.

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 - 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.

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)

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

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 — User's 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.

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

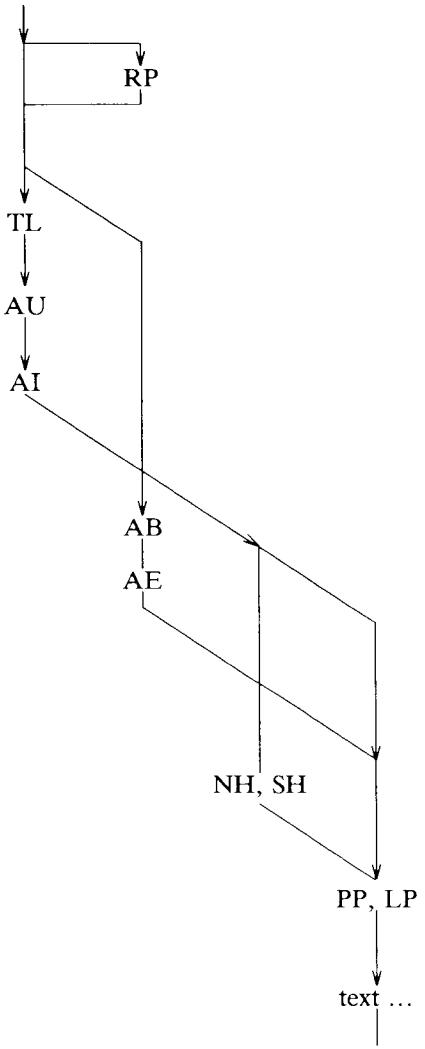


Figure 10-1. Order of Commands in Input

Commands for a TM

Input:

```
.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.

Output:

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**
X. Y. Hardwired	**MH 1K-222**	**5432**

Charging Case- **99999**
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

Input:

```
.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 ...
```

Output:

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

$$\sum_{0}^{\infty} F(x_i) = G(x) \qquad (1)$$

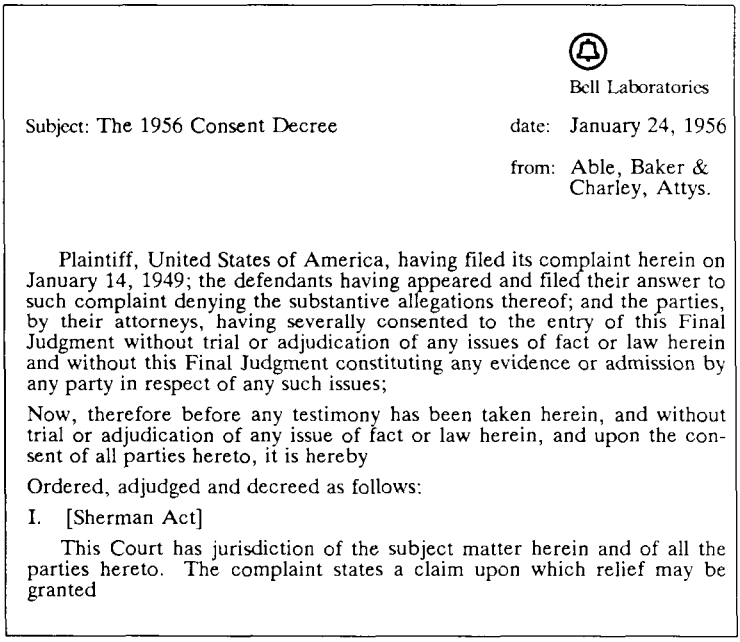
is found with the transformation $x = \frac{\rho}{\theta}$ where $\rho = G'(x)$ and θ is derived from well-known principles.

An Internal Memorandum

Input:

```
.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...
```

Output:



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

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

Input:

```
.NH                          .SH
Introduction.                Appendix I
.PP                          .PP
text text text               text text text
```

Output:

1. Introduction
text text text

Appendix I
text text text

A Simple List

Input:

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

Output:

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

Displays

Input:

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

Output:

hoboken harrison newark roseville avenue grove street east orange brick church orange highland avenue mountain station

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

Input:

```
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
```

Output:

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

Multiple Indents

Input:

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

Output:

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

1. First level item

   a) Second level.

   b) Continued here with another second level item, but somewhat longer.

2. Return to previous value of the indenting at this point.

3. Another line.

---

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

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

Input:

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

Output:

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

Input:

```
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
```

Output:

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

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

Input:

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

Output:

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

Input:

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

Output:

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

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

Input:

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

Output:

$\dot{a}$, $\ddot{b}$, $\tilde{\xi} \times \vec{y}$.

See also the equations in the table example.

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

Input:

```
.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)
```

Output:

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

* (first quarter only)

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

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

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

Input:

```
.TS          (with delim $$ on)
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
```

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	$\zeta(s)=\sum_{k=1}^{\infty}k^{-s}\quad(\mathrm{Re}\ s>1)$

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
```

11

Writing Papers with NROFF Using –me

Introduction

This document describes the text processing facilities available on the UNIX operating system via NROFF and the –me macro package. It is assumed that the reader already is generally familiar with the UNIX operating system and a text editor such as **ex**. This is intended to be a casual introduction, and as such not all material is covered. In particular, many variations and additional features of the –me macro package are not explained. For a complete discussion of this and other issues, see *The –me Reference Manual* and *The NROFF/TROFF Reference Manual.*

NROFF, a computer program that runs on the UNIX operating system, reads an input file prepared by the user and outputs a formatted paper suitable for publication or framing. The input consists of *text*, or words to be printed, and *requests*, which give instructions to the NROFF program telling how to format the printed copy.

Section 1 describes the basics of text processing. Section 2 describes the basic requests. Section 3 introduces displays. Annotations, such as footnotes, are handled in section 4. The more complex requests which are not discussed in section 2 are covered in section 5. Finally, section 6 discusses things you will need to know if you want to typeset documents. If you are a novice, you probably won't want to read beyond section 4 until you have tried some of the basic features out.

When you have your raw text ready, call the NROFF formatter by typing as a request to the UNIX shell:

nroff –me –T*type files*

Source: Eric P. Allman, *Writing Papers with NROFF Using –me* (Berkeley, CA: University of California, 1980).

where *type* describes the type of terminal you are outputting to. Common values are **dtc** for a DTC 300s (daisy-wheel type) printer and **lpr** for the line printer. If the **-T** flag is omitted, a "lowest common denominator" terminal is assumed; this is good for previewing output on most terminals. A complete description of options to the NROFF command can be found in *The NROFF/TROFF Reference Manual*.

The word *argument* is used in this manual to mean a word or number which appears on the same line as a request which modifies the meaning of that request. For example, the request

```
.sp
```

spaces one line, but

```
.sp 4
```

spaces four lines. The number **4** is an *argument* to the **.sp** request which says to space four lines instead of one. Arguments are separated from the request and from each other by spaces.

1. Basics of Text Processing

The primary function of NROFF is to *collect* words from input lines, *fill* output lines with those words, *justify* the right hand margin by inserting extra spaces in the line, and output the result. For example, the input:

```
Now is the time
for all good men
to come to the aid
of their party.
Four score and seven
years ago,...
```

will be read, packed onto output lines, and justified to produce:

```
Now is the time for all good men to come to the aid of their
party.  Four score and seven years ago,...
```

Sometimes you may want to start a new output line even though the line you are on is not yet full; for example, at the end of a paragraph.

To do this you can cause a *break*, which starts a new output line. Some requests cause a break automatically, as do blank input lines and input lines beginning with a space.

Not all input lines are text to be formatted. Some of the input lines are *requests* which describe how to format the text. Requests always have a period or an apostrophe (" ' ") as the first character of the input line.

The text formatter also does more complex things, such as automatically numbering pages, skipping over page folds, putting footnotes in the correct place, and so forth.

I can offer you a few hints for preparing text for input to NROFF. First, keep the input lines short. Short input lines are easier to edit, and NROFF will pack words onto longer lines for you anyhow. In keeping with this, it is helpful to begin a new line after every period, comma, or phrase, since common corrections are to add or delete sentences or phrases. Second, do not put spaces at the end of lines, since this can sometimes confuse the NROFF processor. Third, do not hyphenate words at the end of lines (except words that should have hyphens in them, such as "mother-in-law"); NROFF is smart enough to hyphenate words for you as needed, but is not smart enough to take hyphens out and join a word back together. Also, words such as "mother-in-law" should not be broken over a line, since then you will get a space where not wanted, such as "mother- in-law".

2. Basic Requests

2.1 Paragraphs

Paragraphs are begun by using the **.pp** request. For example, the input:

```
.pp
Now is the time for all good men
to come to the aid of their party.
Four score and seven years ago,...
```

produces a blank line followed by an indented first line. The result is:

> Now is the time for all good men to come to the aid of their party. Four score and seven years ago,...

Notice that the sentences of the paragraphs *must not* begin with a space, since blank lines and lines beginning with spaces cause a break. For example, if I had typed:

```
.pp
Now is the time for all good men
    to come to the aid of their party.
Four score and seven years ago,...
```

The output would be:

> Now is the time for all good men
> to come to the aid of their party. Four score and seven years ago,...

A new line begins after the word "men" because the second line began with a space character.

There are many fancier types of paragraphs, which will be described later.

2.2 Headers and Footers

Arbitrary headers and footers can be put at the top and bottom of every page. Two requests of the form **.he** *title* and **.fo** *title* define the titles to put at the head and the foot of every page, respectively. The titles are called *three-part* titles, that is, there is a left-justified part, a centered part, and a right-justified part. To separate these three parts the first character of *title* (whatever it may be) is used as a delimiter. Any character may be used, but backslash and double quote marks should be avoided. The percent sign is replaced by the current page number whenever found in the title. For example, the input:

```
.he ´´%´´
.fo ´Jane Jones´´My Book´
```

results in the page number centered at the top of each page, "Jane Jones" in the lower left corner, and "My Book" in the lower right corner.

2.3 Double Spacing

NROFF will double space output text automatically if you use the request **.ls 2**, as is done in this section. You can revert to single spaced mode by typing **.ls 1**.

2.4 Page Layout

A number of requests allow you to change the way the printed copy looks, sometimes called the *layout* of the output page. Most of these requests adjust the placing of "white space" (blank lines or spaces). In these explanations, characters in italics should be replaced with values you wish to use; bold characters represent characters which should actually be typed.

The **.bp** request starts a new page.

The request **.sp** *N* leaves *N* lines of blank space. *N* can be omitted (meaning skip a single line) or can be of the form *N***i** (for *N* inches) or *N***c** (for *N* centimeters). For example, the input:

```
.sp 1.5i
My thoughts on the subject
.sp
```

leaves one and a half inches of space, followed by the line "My thoughts on the subject," followed by a single blank line.

The **.in** +*N* request changes the amount of white space on the left of the page (the *indent*). The argument *N* can be of the form +*N* (meaning leave *N* spaces more than you are already leaving), –*N* (meaning leave less than you do now), or just *N* (meaning leave exactly *N* spaces). *N* can be of the form *N***i** or *N***c** also. For example, the input:

```
initial text
.in 5
some text
.in +1i
more text
.in -2c
final text
```

produces "some text" indented exactly five spaces from the left

margin, "more text" indented five spaces plus one inch from the left margin (fifteen spaces on a pica typewriter), and "final text" indented five spaces plus one inch minus two centimeters from the margin. That is, the output is:

```
initial text
          some text
                        more text
            final text
```

The **.ti** *+N* (temporary indent) request is used like **.in** *+N* when the indent should apply to one line only, after which it should revert to the previous indent. For example, the input:

```
.in 1i
.ti 0
Ware, James R.  The Best of Confucius,
Halcyon House, 1950.
An excellent book containing translations of
most of Confucius' most delightful sayings.
A definite must for anyone interested in the early foundations
of Chinese philosophy.
```

produces:

Ware, James R. The Best of Confucius, Halcyon House, 1950. An excellent book containing translations of most of Confucius' most delightful sayings. A definite must for anyone interested in the early foundations of Chinese philosophy.

Text lines can be centered by using the **.ce** request. The line after the **.ce** is centered (horizontally) on the page. To center more than one line, use **.ce** *N* (where *N* is the number of lines to center), followed by the *N* lines. If you want to center many lines but don't want to count them, type:

```
.ce 1000
lines to center
.ce 0
```

The **.ce 0** request tells NROFF to center zero more lines, in other words, stop centering.

All of these requests cause a break; that is, they always start a new line. If you want to start a new line without performing any other action, use **.br**.

2.5 Underlining

Text can be underlined using the **.ul** request. The **.ul** request causes the next input line to be underlined when output. You can underline multiple lines by stating a count of *input* lines to underline, followed by those lines (as with the **.ce** request). For example, the input:

```
.ul 2
Notice that these two input lines
are underlined.
```

will underline those eight words in NROFF. (In TROFF they will be set in italics.)

3. Displays

Displays are sections of text to be set off from the body of the paper. Major quotes, tables, and figures are types of displays, as are all the examples used in this document. All displays except centered blocks are output single spaced.

3.1 Major Quotes

Major quotes are quotes which are several lines long, and hence are set in from the rest of the text without quote marks around them. These can be generated using the commands **.(q** and **.)q** to surround the quote. For example, the input:

```
As Weizenbaum points out:
.(q
It is said that to explain is to explain away.
This maxim is nowhere so well fulfilled
as in the areas of computer programming,...
.)q
```

generates as output:

As Weizenbaum points out:

> It is said that to explain is to explain away. This maxim is nowhere so well fulfilled as in the areas of computer programming,...

3.2 Lists

A *list* is an indented, single spaced, unfilled display. Lists should be used when the material to be printed should not be filled and justified like normal text, such as columns of figures or the examples used in this paper. Lists are surrounded by the requests **.(l** and **.)l**. For example, type:

```
Alternatives to avoid deadlock are:
.(l
Lock in a specified order
Detect deadlock and back out one process
Lock all resources needed before proceeding
.)l
```

will produce:

Alternatives to avoid deadlock are:

Lock in a specified order
Detect deadlock and back out one process
Lock all resources needed before proceeding

3.3 Keeps

A *keep* is a display of lines which are kept on a single page if possible. An example of where you would use a keep might be a diagram. Keeps differ from lists in that lists may be broken over a page boundary whereas keeps will not.

Blocks are the basic kind of keep. They begin with the request **.(b** and end with the request **.)b**. If there is not room on the current page for everything in the block, a new page is begun. This has the unpleasant effect of leaving blank space at the bottom of the page. When this is not appropriate, you can use the alternative, called *floating keeps*.

Floating keeps move relative to the text. Hence, they are good for things which will be referred to by name, such as "See figure 3" . A floating keep will appear at the bottom of the current page if it will fit; otherwise, it will appear at the top of the next page. Floating keeps begin with the line **.(z** and end with the line **.)z**. For an example of a floating keep, see figure 1. The **.hl** request is used to draw a horizontal line so that the figure stands out from the text.

```
.(z
.hl
Text of keep to be floated.
.sp
.ce
Figure 1.  Example of a Floating Keep.
.hl
.)z
```

Figure 1. Example of a Floating Keep.

3.4 Fancier Displays

Keeps and lists are normally collected in *nofill* mode, so that they are good for tables and such. If you want a display in fill mode (for text), type **.(l F** (Throughout this section, comments applied to **.(l** also apply to **.(b** and **.(z**). This kind of display will be indented from both margins. For example, the input:

```
.(l F
And now boys and girls,
a newer, bigger, better toy than ever before!
Be the first on your block to have your own computer!
Yes kids, you too can have one of these modern
data processing devices.
You too can produce beautifully formatted papers
without even batting an eye!
.)l
```

will be output as:

> And now boys and girls, a newer, bigger, better toy than ever before! Be the first on your block to have your own computer! Yes kids, you too can have one of these modern data processing devices. You too can produce beautifully formatted papers without even batting an eye!

Lists and blocks are also normally indented (floating keeps are normally left justified). To get a left-justified list, type **.(l L**. To get a list centered line-for-line, type **.(l C**. For example, to get a filled, left justified list, enter:

```
.(l L F
text of block
.)l
```

The input:

```
.(l
first line of unfilled display
more lines
.)l
```

produces the indented text:

first line of unfilled display
more lines

Typing the character **L** after the **.(l** request produces the left justified result:

first line of unfilled display
more lines

Using **C** instead of **L** produces the line-at-a-time centered output:

first line of unfilled display
more lines

Sometimes it may be that you want to center several lines as a group, rather than centering them one line at a time. To do this use centered blocks, which are surrounded by the requests **.(c** and **.)c**. All the lines are centered as a unit, such that the longest line is centered and the rest are lined up around that line. Notice that lines do not move relative to each other using centered blocks, whereas they do using the **C** argument to keeps.

Centered blocks are *not* keeps, and may be used in conjunction with keeps. For example, to center a group of lines as a unit and keep them on one page, use:

```
.(b L
.(c
first line of unfilled display
more lines
.)c
.)b
```

to produce:

first line of unfilled display
more lines

If the block requests (**.(b** and **.)b**) had been omitted the result would have been the same, but with no guarantee that the lines of the centered block would have all been on one page. Note the use of the **L** argument to **.(b**; this causes the centered block to center within the entire line rather than within the line minus the indent. Also, the center requests must be nested *inside* the keep requests.

4. Annotations

There are a number of requests to save text for later printing. *Footnotes* are printed at the bottom of the current page. *Delayed text* is intended to be a variant form of footnote; the text is printed only when explicitly called for, such as at the end of each chapter. *Indexes* are a type of delayed text having a tag (usually the page number) attached to each entry after a row of dots. Indexes are also saved until called for explicitly.

4.1 Footnotes

Footnotes begin with the request **.(f** and end with the request **.)f**. The current footnote number is maintained automatically, and can be used by typing **, to produce a footnote number[1]. The number is automatically incremented after every footnote. For example, the input:

1. Like this.

```
.(q
A man who is not upright
and at the same time is presumptuous;
one who is not diligent and at the same time is ignorant;
one who is untruthful and at the same time is incompetent;
such men I do not count among acquaintances.\**
.(f
\**James R. Ware,
.ul
The Best of Confucius,
Halcyon House, 1950.
Page 77.
.)f
.)q
```

generates the result:

> A man who is not upright and at the same time is presumptuous; one who is not diligent and at the same time is ignorant; one who is untruthful and at the same time is incompetent; such men I do not count among acquaintances.[2]

It is important that the footnote appears *inside* the quote, so that you can be sure that the footnote will appear on the same page as the quote.

4.2 Delayed Text

Delayed text is very similar to a footnote except that it is printed when called for explicitly. This allows a list of references to appear (for example) at the end of each chapter, as is the convention in some disciplines. Use *# on delayed text instead of ** as on footnotes.

If you are using delayed text as your standard reference mechanism, you can still use footnotes, except that you may want to reference them with special characters* rather than numbers.

---

2. James R. Ware, *The Best of Confucius*, Halcyon House, 1950. Page 77.
* Such as an asterisk.

4.3 Indexes

An "index" (actually more like a table of contents, since the entries are not sorted alphabetically) resembles delayed text, in that it is saved until called for. However, each entry has the page number (or some other tag) appended to the last line of the index entry after a row of dots.

Index entries begin with the request **.(x** and end with **.)x**. The **.)x** request may have a argument, which is the value to print as the "page number". It defaults to the current page number. If the page number given is an underscore ("_") no page number or line of dots is printed at all. To get the line of dots without a page number, type **.)x "" ** , which specifies an explicitly null page number. The **.xp** request prints the index.

For example, the input:

```
.(x
Sealing wax
.)x
.(x
Cabbages and kings
.)x _
.(x
Why the sea is boiling hot
.)x 2.5a
.(x
Whether pigs have wings
.)x ""
.(x
This is a terribly long index entry, such as might be used
for a list of illustrations, tables, or figures; I expect it to
take at least two lines.
.)x
.xp
```

generates:

The **.(x** request may have a single character argument, specifying the "name" of the index; the normal index is **x**. Thus, several "indices" may be maintained simultaneously (such as a list of tables, table of contents, etc.).

Notice that the index must be printed at the *end* of the paper, rather than at the beginning where it will probably appear (as a table of contents); the pages may have to be physically rearranged after printing.

5. Fancier Features

A large number of fancier requests exist, notably requests to provide other sorts of paragraphs, numbered sections of the form **1.2.3** (such as used in this document), and multicolumn output.

5.1 More Paragraphs

Paragraphs generally start with a blank line and with the first line indented. It is possible to get left-justified block-style paragraphs by using **.lp** instead of **.pp**, as demonstrated by the next paragraph.

Sometimes you want to use paragraphs that have the *body* indented, and the first line exdented (opposite of indented) with a label. This can be done with the **.ip** request. A word specified on the same line as **.ip** is printed in the margin, and the body is lined up at a prespecified position (normally five spaces). For example, the input:

```
.ip one
This is the first paragraph.
Notice how the first line
of the resulting paragraph lines up
with the other lines in the paragraph.
.ip two
And here we are at the second paragraph already.
You may notice that the argument to .ip
appears
in the margin.
.lp
We can continue text...
```

produces as output:

one This is the first paragraph. Notice how the first line of the resulting paragraph lines up with the other lines in the paragraph.

two And here we are at the second paragraph already. You may notice that the argument to **.ip** appears in the margin.

We can continue text without starting a new indented paragraph by using the **.lp** request.

If you have spaces in the label of a **.ip** request, you must use an "unpaddable space" instead of a regular space. This is typed as a backslash character ("\") followed by a space. For example, to print the label "Part 1", enter:

```
.ip "Part\ 1"
```

If a label of an indented paragraph (that is, the argument to **.ip**) is longer than the space allocated for the label, **.ip** will begin a new line after the label. For example, the input:

```
.ip longlabel
This paragraph had a long label.
The first character of text on the first line
will not line up with the text on second and subsequent lines,
although they will line up with each other.
```

will produce:

longlabel
This paragraph had a long label. The first character of text on the first line will not line up with the text on second and subsequent lines, although they will line up with each other.

It is possible to change the size of the label by using a second argument which is the size of the label. For example, the above example could be done correctly by saying:

```
.ip longlabel 10
```

which will make the paragraph indent 10 spaces for this paragraph only. If you have many paragraphs to indent all the same amount, use the *number register* **ii**. For example, to leave one inch of space for the label, type:

```
.nr ii 1i
```

somewhere before the first call to **.ip**. Refer to the reference manual for more information.

If **.ip** is used with no argument at all no hanging tag will be printed. For example, the input:

```
.ip [a]
This is the first paragraph of the example.
We have seen this sort of example before.
.ip
This paragraph is lined up with the previous paragraph,
but it has no tag in the margin.
```

produces as output:

[a] This is the first paragraph of the example. We have seen this sort of example before.

This paragraph is lined up with the previous paragraph, but it has no tag in the margin.

A special case of **.ip** is **.np**, which automatically numbers paragraphs sequentially from 1. The numbering is reset at the next **.pp**, **.lp**, or **.sh** (to be described in the next section) request. For example, the input:

```
.np
This is the first point.
.np
This is the second point.
Points are just regular paragraphs
which are given sequence numbers automatically
by the .np request.
.pp
This paragraph will reset numbering by .np.
.np
For example,
we have reverted to numbering from one now.
```

generates:

(1) This is the first point.

(2) This is the second point. Points are just regular paragraphs which are given sequence numbers automatically by the .np request.

This paragraph will reset numbering by .np.

(1) For example, we have reverted to numbering from one now.

The **.bu** request gives lists of this sort that are identified with bullets rather than numbers. The paragraphs are also crunched together.

For example, the input:

```
.bu
One egg yolk
.bu
One tablespoon cream or top milk
.bu
Salt, cayenne, and lemon juice to taste
.bu
A generous two tablespoonfuls of butter
```

produces[3]:

- One egg yolk
- One tablespoon cream or top milk
- Salt, cayenne, and lemon juice to taste
- A generous two tablespoonfuls of butter

5.2 Section Headings

Section numbers (such as the ones used in this document) can be automatically generated using the **.sh** request. You must tell **.sh** the *depth* of the section number and a section title. The depth specifies how many numbers are to appear (separated by decimal points) in the section number. For example, the section number **4.2.5** has a depth of three.

Section numbers are incremented in a fairly intuitive fashion. If you add a number (increase the depth), the new number starts out at one. If you subtract section numbers (or keep the same number) the final number is incremented. For example, the input:

3. By the way, if you put the first three ingredients in a a heavy, deep pan and whisk the ingredients madly over a medium flame (never taking your hand off the handle of the pot) until the mixture reaches the consistency of custard (just a minute or two), then mix in the butter off-heat, you will have a wonderful Hollandaise sauce.

```
.sh 1 "The Preprocessor"
.sh 2 "Basic Concepts"
.sh 2 "Control Inputs"
.sh 3
.sh 3
.sh 1 "Code Generation"
.sh 3
```

produces as output the result:

1. The Preprocessor
1.1. Basic Concepts
1.2. Control Inputs
1.2.1.
1.2.2.
2. Code Generation
2.1.1.

You can specify the section number to begin by placing the section number after the section title, using spaces instead of dots. For example, the request:

```
.sh 3 "Another section" 7 3 4
```

will begin the section numbered **7.3.4**; all subsequent **.sh** requests will number relative to this number.

There are more complex features which will cause each section to be indented proportionally to the depth of the section. For example, if you enter:

```
.nr si N
```

each section will be indented by an amount *N*. *N* must have a scaling factor attached, that is, it must be of the form *Nx*, where *x* is a character telling what units *N* is in. Common values for *x* are **i** for inches, **c** for centimeters, and **n** for *ens* (the width of a single character). For example, to indent each section one-half inch, type:

```
.nr si 0.5i
```

After this, sections will be indented by one-half inch per level of depth in the section number.

Section headers without automatically generated numbers can be done using:

```
.uh "Title"
```

which will do a section heading, but will put no number on the section.

5.3 Parts of the Basic Paper

There are some requests which assist in setting up papers. The **.tp** request initializes for a title page. There are no headers or footers on a title page, and unlike other pages you can space down and leave blank space at the top. For example, a typical title page might appear as:

```
.tp
.sp 2i
.(l C
THE GROWTH OF TOENAILS
IN UPPER PRIMATES
.sp
by
.sp
Frank N. Furter
.)l
.bp
```

12

Typesetting Mathematics — User's Guide (Second Edition)

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.

Source: Brian W. Kernighan and Lorinda L. Cherry, *Typesetting Mathematics — User's Guide (Second Edition)* (Murray Hill, N.J.: Bell Laboratories, 1978).

The same language may be used with the UNIX formatter NROFF to set mathematical expressions on DASI and GSI terminals and Model 37 teletypes.

1. Introduction

EQN is a program for typesetting mathematics on the Graphics Systems phototypesetters on the UNIX operating system. 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
```

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 I 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 *xsub*2 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}}
```

$$e^{i\,\pi^{\rho+1}}$$

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

$$\sqrt{a+b} + \frac{1}{\sqrt{ax^2+bx+c}}$$

CAUTION

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:

$$(a^2/b_2)^{½}$$

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,

$$\int \quad \prod \quad \cup \quad \cap$$

Since the thing before the *from* can be anything, even something in braces, *from-to* can often be used in unexpected ways:

```
lim from {n -> inf} x sub n =0
```

is

$$\lim_{n \to \infty} x_n = 0$$

12. Size and Font Changes

By default, equations are set in 10-point type with standard mathematical conventions to determine what characters are in roman and what in italic. Although EQN makes a valiant attempt to use 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	$\vec{\vec{x}}$
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

$$sin(x) + \sin(x)$$

‡ 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.

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}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

```
.EQ I
x+y mark = z
.EN
.EQ I
x lineup = 1
.EN
```

to produce

$$x+y=z$$
$$x=1$$

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

$$\left\lfloor \frac{x}{y} \right\rfloor \leq \left\lceil \frac{a}{b} \right\rceil$$

Several warnings about brackets are in order. First, braces are typically bigger than brackets and parentheses, because they are made up of three, five, seven, etc., pieces, while brackets can be made up of two, three, etc. Second, big left and right parentheses often look poor, because the character set is poorly designed.

The *right* part may be omitted: a "left something" need not have a corresponding "right something". If the *right* part is omitted, put braces around the thing you want the left bracket to encompass. Otherwise, the resulting brackets may be too large.

If you want to omit the *left* part, things are more complicated, because technically you can't have a *right* without a corresponding *left*. Instead you have to say

```
left "" ..... right )
```

for example. The *left* "" means a "left nothing". This satisfies the rules without hurting your output.

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

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Notice the left brace without a matching right one.

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
```

CAUTION

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} } ove {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.

>=	≥	approx	≈
<=	≤	nothing	
==	≡	cdot	·
!=	≠	times	×
+-	±	del	∇
->	→	grad	∇
<-	←	...	. . .
<<	≪	,....,	, ... ,
>>	≫	sum	Σ
inf	∞	int	∫
partial	∂	prod	Π
half	½	union	∪
prime	′	inter	∩

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	ο
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* 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
```

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 -Tx
```

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.

13

Tbl — A Program to Format Tables

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

Source: M. E. Lesk, *Tbl – A Program to Format Tables* (Murray Hill, N.J.: Bell Laboratories, 1979).

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 21);

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.

CAUTION

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 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.

* More precisely, an en is a number of points (1 point = 1/72 inch) equal to half the current type size.

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 *tabls* 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.

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 17 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

CAUTION

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 14 and 21. Using this procedure, each table line can be close to its corresponding format line.

CAUTION

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 ① in the input represents a tab character.

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
box;
c c c
l l l.
Language ① Authors ① Runs on

Fortran ① Many ① Almost anything
PL/1 ① IBM ① 360/370
C ① BTL ① 11/45,H6000,370
BLISS ① Carnegie-Mellon ① PDP-10,11
IDS ① Honeywell ① H6000
Pascal ① Stanford ① 370
.TE
```

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;
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

Output:

Major New York Bridges		
Bridge	Designer	Length
Brooklyn	J. A. Roebling	1595
Manhattan	G. Lindenthal	1470
Williamsburg	L. L. Buck	1600
Queensborough	Palmer & Hornbostel	1182
Triborough	O. H. Ammann	1380
		383
Bronx Whitestone	O. H. Ammann	2300
Throgs Neck	O. H. Ammann	1800
George Washington	O. H. Ammann	3500

Input:

```
.TS
box;
c s s
c | c | c
l | l | n.
Major New York Bridges
=
Bridge ⓣ Designer ⓣ Length
_
Brooklyn ⓣ J. A. Roebling ⓣ 1595
Manhattan ⓣ G. Lindenthal ⓣ 1470
Williamsburg ⓣ L. L. Buck ⓣ 1600
_
Queensborough ⓣ Palmer & ⓣ 1182
ⓣ Hornbostel
_
ⓣ ⓣ 1380
Triborough ⓣ O. H. Ammann ⓣ _
ⓣ ⓣ 383
_
Bronx Whitestone ⓣ O. H. Ammann ⓣ 2300
Throgs Neck ⓣ O. H. Ammann ⓣ 1800
_
George Washington ⓣ O. H. Ammann ⓣ 3500
.TE
```

Input:

```
.TS
c c
np-2 | n | .
⓽Stack
⓽_
1 ⓽46
⓽_
2 ⓽23
⓽_
3 ⓽15
⓽_
4 ⓽6.5
⓽_
5 ⓽2.1
⓽_
.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 ⓽february ⓽march
april ⓽may
june ⓽july ⓽Months
august ⓽september
october ⓽november ⓽december
.TE
```

Output:

january	february	march
april	may	
june	july	**Months**
august	september	
october	november	december

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:

```
.TS
allbox;
cfI s s
c  cw(1i)  cw(1i)
lp10 lp10 lp10.
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
```

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} d
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	$\zeta(s)=\sum_{k=1}^{\infty} k^{-s}$ (Re $s>1$)

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:

```
.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:

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
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:

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:

```
.TS
c s
cip-2 s
l n
a n.
Some London Transport Statistics
(Year 1964)
Railway route miles⓽244
Tube⓽66
Sub-surface⓽22
Surface⓽156
.sp .5
.T&
l r
a r.
Passenger traffic \- railway
Journeys⓽674 million
Average length⓽4.55 miles
Passenger miles⓽3,066 million
.T&
l r
a r.
Passenger traffic \- road
Journeys⓽2,252 million
Average length⓽2.26 miles
Passenger miles⓽5,094 million
.T&
l n
a n.
.sp .5
Vehicles⓽12,521
Railway motor cars⓽2,905
Railway trailer cars⓽1,269
Total railway⓽4,174
Omnibuses⓽8,347
.T&
l n
a n.
.sp .5
Staff⓽73,739
Administrative, etc.⓽5,582
Civil engineering⓽5,134
Electrical eng.⓽1,714
Mech. eng. \- railway⓽4,310
Mech. eng. \- road⓽9,152
Railway operations⓽8,930
Road operations⓽35,946
Other⓽2,971
.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 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. 718-638-5000
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

Input:

```
.TS
box;
cb  s  s  s
c | c | c  s
ltiw(.75i) | ltw(1.5i) | lp8 | lw(1.25i)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 ⓣ 718-638-5000

_
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:

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
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
```

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 J. F. Ossanna, whose assistance with this program in particular has 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*, Bell Laboratories internal memorandum.

[5] M. E. Lesk and B. W. Kernighan, *Computer Typesetting of Technical Journals on UNIX*, Computing Science Technical Report No. 44, Bell Laboratories, July 1976.

[6] J. R. Mashey and D. W. Smith, *PWB/MM — Programmer's Workbench Memorandum Macros*, Bell Laboratories memorandum.

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

14

S/1280 Inter-CPU Communication

The *Inter-CPU Communication* facility (ICC) provides communication between programs running on S/1280's various CPUs. The system kernels running on the various Application Processors use ICC to coordinate with each other and with CTOS; this permits these kernels to act as a single CTIX kernel.

CTIX processes can use ICC to communicate with each other or with CTOS system services. If both processes participating are CTIX processes, ICC is simply an enhanced communication facility and it does not matter whether the two processes actually run on different CPUs. If one of the processes provides a CTOS system service, ICC is a means to use CTOS controlled resources directly, without working through the CTIX kernel.

Processes need no special status or permissions to use ICC. Each ICC action is executed by a CTIX system call.

The important concept behind ICC is that processes provide other processes *services*. For example, a database service permits the storage and retrieval of records; a spooling service queues files.

ICC labels CTIX processes by the roles they play in ICC communications. A *server* is a process that provides a service; a *client* is a process that uses a service. For example, users might run query programs to specify retrievals. The query programs send messages to a database management service; the program in charge of the database does the actual retrieval. The query programs are clients of the database management service; the retrieval program is the server for the database management service. These labels are only valid in reference to a particular service: a server for one service might be a client of another service.

Each ICC interaction consists of two messages. A *request* is a message from a client to a server. A *response* is a return message from a server to a client. A server must provide a response to each request.

Clients and servers do not address each other directly. A client specifies that a message is a request to a specific service; the kernel sends the request to whatever server provides that service. A server specifies that a message is a response to a specific request; the kernel sends the response the sender of that request.

Points of View

When a process uses ICC it plays the role of client or server. This role affects the way the process views ICC. Here we summarize the points of view. For the moment, we assume a process has but one point of view.

The Client

There are actually two kinds of client. The *basic* client uses a simplified system call that is adequate for most uses. The *general* client has a complicated repertoire of system calls.

A basic client sends a single message and waits for its response. This action requires a single system call that dispatches the request, waits for the response, and copies the response (Figure 14–1).

A general client can send any number of messages at a time. It manages the resulting responses with one or more message queues (Figure 14–2). The client must periodically examine these queues and delete responses from them.

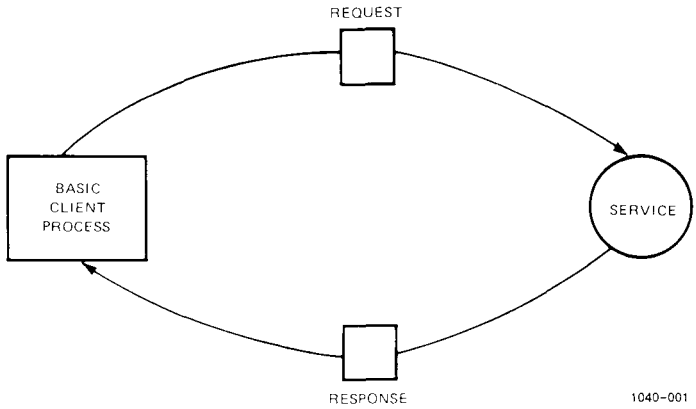


Figure 14-1. A Basic Client

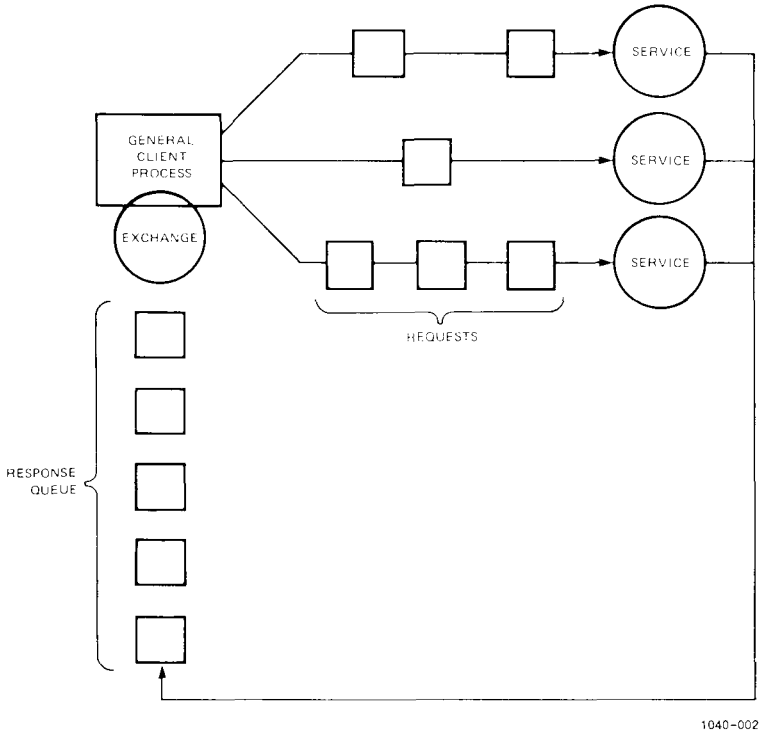


Figure 14-2. A General Client

A basic client actually possesses a response queue just like that of a general client. The kernel manages the queue on the client's behalf.

The Server

A server manages its incoming requests with one or more request queues (Figure 14–3). Every request for a particular service goes to a specific request queue. A server that provides more than one service can use one request queue or several.

A server need not respond to requests in the order it receives them, but it must respond to every request.

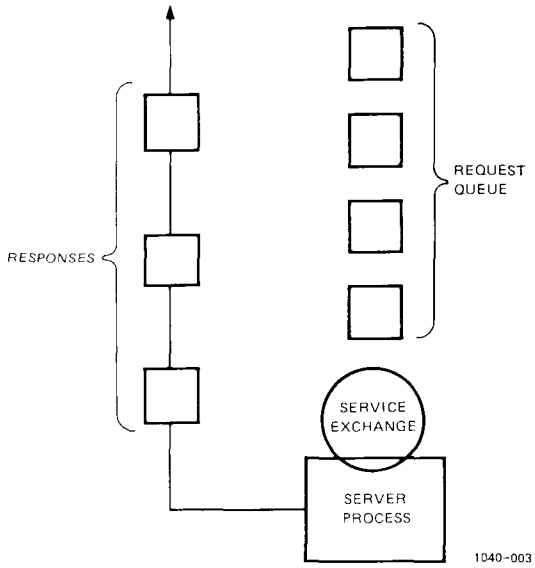


Figure 14-3. A Server

Exchanges

Clients and servers manage their message queues via *exchanges*. One exchange handles each queue. When a client sends a request, it tells the kernel to queue the resulting response at a specific exchange. When a server wants to provide a service, it tells the kernel to queue requests for that service at a specific exchange. A process specifies an exchange when it asks the kernel for newly queued messages.

A Note on Source Code

This section mentions several standard structure types and constants. CTIX provides a C version of these definitions in **/usr/include/exch.h**. In a C ICC program, include this file with

```
#include <exch.h>
```

Programmers who work in languages other than C should see the appropriate language manual for definitions of equivalent data structures.

The Basic Client

For the basic client, each ICC interaction has three steps:

- Describing the request with the request block.
- Calling the service.
- Examining the response.

The Request Block

A client describes a request with a data structure called a *request block* (Figure 14–4). ICC requires that each request begin with a standard group of fields called a *request header*. A server can require additional data fields. We will discuss the data fields first.

A request block contains three groups of data fields:

- *Control information* is data that the client sends to the server by placing it in the request block. The control information immediately follows the request header.
- *Request PbCbs* immediately follow the control information. Each request PbCb specifies the beginning of and length of a block of data. This data is sent to the server.
- *Response PbCbs* immediately follow the request PbCbs. Each response PbCb specifies the beginning and length of a block of unused memory. Response data goes into this memory.

A *pointer to bytes/count of bytes* (PbCb) describes a data area outside the request block. The user include file defines a PbCb this way:

```
struct PbCb {
        char *pc_offset;
        unsigned short pc_count;
};
```

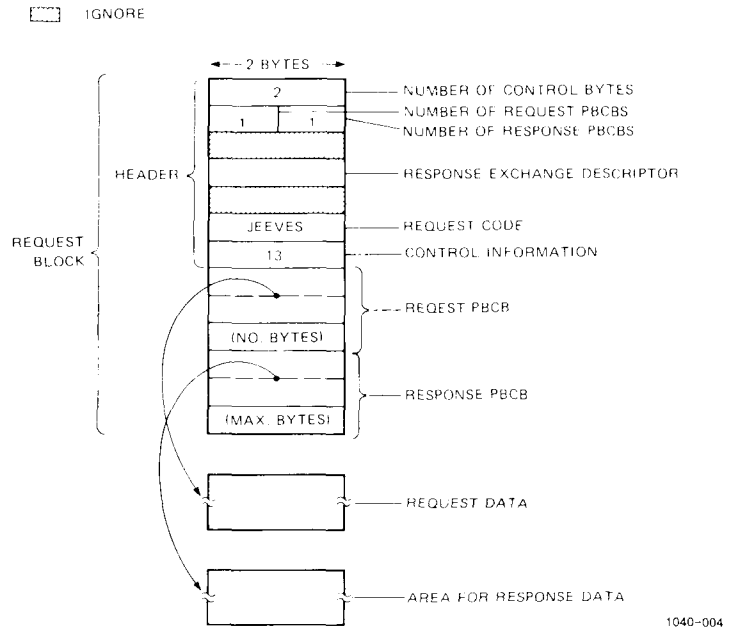


Figure 14-4. A Basic Client's Request to JEEVES

Some servers are rigid as to the number and kind of data fields. A CTOS server, in particular, must always have a specific number of control information bytes, request PbCbs, and response PbCbs. Be sure to follow the data field rules specified for the service.

The include file defines a request header thus:

```
struct rqheader {
        unsigned short r_sCntInfo;
        unsigned char r_nReqPbCb;
        unsigned char r_nRespPbCb;
        unsigned short r_userNum;
        unsigned short r_exchResp;
        unsigned short r_ercRet;
        unsigned short r_rqCode;
};
```

The client does not set **r_userNum** or **r_ercRet**. The other fields require the following values:

r_sCntInfo Number of bytes of control information.

r_nReqPbCb Number of request PbCbs.

r_nRespPbCb Number of response PbCbs.

r_exchResp The *response exchange descriptor*. The definition of this term is not important to a basic client. Set **r_exchResp** to the value returned by **ExQueryDfltRespExch** (no arguments). We will define response exchange descriptor when we get to general clients.

r_rqCode The *request code*. This specifies the service addressed.

Example

Suppose that a service is identified by the constant **JEEVES**. **JEEVES** requires two bytes of control information, one request PbCb, and one response PbCb. A client of **JEEVES** might declare the request block thus:

```
struct jblock {
        struct rqheader jheader;
        short jcinfo;
        struct PbCb reqdata;
        struct PbCb respdata;
} jmess;
```

The client sends 13 in the control data and a string **name** as request data. The client provides an array **respdata** for the response data. The following code sets up the request block:

```
jmess.jheader.r_sCntInfo = 2;
jmess.jheader.r_nReqPbCb = 1;
jmess.jheader.r_nRespPbCb = 1;
jmess.jheader.r_exchResp = exQueryDfltRespExch();
jmess.jheader.r_rqCode = JEEVES;
jmess.jcinfo = 13;
jmess.reqdata.pc_offset = name;
jmess.reqdata.pc_count = strlen(name);
jmess.respdata.pc_offset = respdata;
jmess.respdata.pc_count = sizeof(respdata);
```

Call the Service

ExCall sends a request block to a service and waits for a response. This system call is defined thus:

```
exCall(reqblp);
struct rqheader *reqblp;
```

Reqblp must point to the request block that describes the request. If the kernel succeeds in sending the request and obtaining a response, **exCall** returns 0. If the kernel cannot send the message, **exCall** returns -1 and sets the external variable **errno**.

Example

This call passes sends the JEEVES request block:

```
if (exCall(&jmess) == -1) {
        perror("can't send message");
exit(1);
}
```

The Modified Request Block

A successful use of **exCall** modifies the request block. This provides the error return code and response data from the server:

- **R_ercRet** in the request header contains the server's return code. Conventionally, a zero return code means that the server performed the client's service.
- Each region of memory described by a response PbCb contains response data from the server. If a server provides more data than the client allowed for, the kernel right-truncates the server data to fit. ICC conventions do not provide a way for the server to tell the client the amount of response data actually sent; the server's own conventions must provide for this, if necessary.

Figure 14–5 shows how a response from JEEVES modifies the client's request block.

The General Client

The general client takes separate action to issue requests and to receive responses. This section describes requests and responses as separate issues—but remember that each request produces a response.

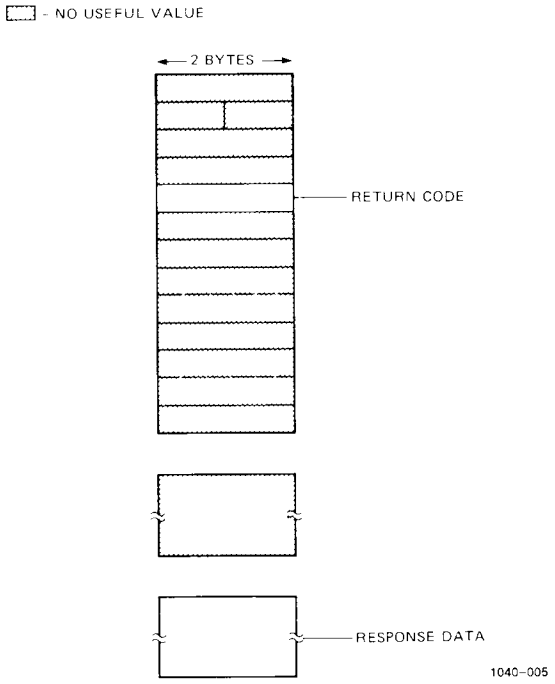


Figure 14-5. Request Block Modified by Response From JEEVES

The Request Side

The general client describes its request with a request block. This request block has the same format as a request block used by a basic client, but two fields change in function:

- **R_exchResp** in the request header. This must be an *exchange descriptor*. If a client only uses one response queue, it can use the exchange descriptor returned by **exQueryDfltRespExch**. If a client uses more than one response queue, it must allocate additional exchanges with **exAllocExch** (no arguments). The kernel will queue the response to this request at the exchange indicated by **r_exchResp**.

- The client need not specify the **pc_offset** fields in the response PbCbs: the kernel ignores them. It is still important to put the data area sizes in the **pc_count** fields: the kernel passes this data to the server.

Figure 14–6 shows a general client's request to **JEEVES**.

The general client sends requests with **exRequest**, which is defined thus:

```
unsigned short exRequest(reqblp);
struct rqheader *reqblp;
```

As with **exCall**, **reqblp** points to a request block that describes the request.

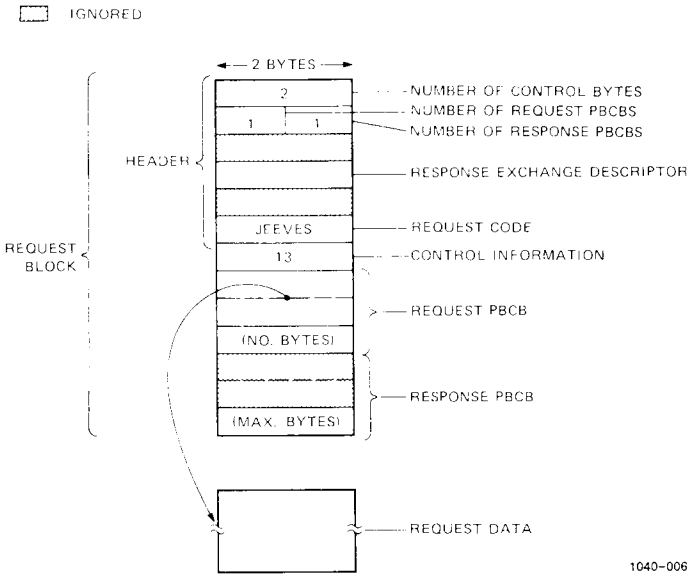


Figure 14-6. A General Client's Request to **JEEVES**

ExRequest returns a *request descriptor*. The request descriptor uniquely identifies the request and its response in subsequent dialogues between the client and the kernel.

ExRequest copies the request block before it returns. Thus the client can safely modify or deallocate the request block once **ExRequest** returns.

Example

The general client version of our **JEEVES** example uses the following call:

```
if (rqdes1 = exRequest(&jmess) == RQDES_INVALID) {
        perror("can't send message");
        exit(1);
}
```

This call copies **jmess** and the request data area and sends them to the server. It leaves **jmess** unaltered. The ICC include file defines **RQDES_INVALID**, a value that cannot be a valid request descriptor.

The Response Side

Responses queue up at the client's exchanges in the order the servers send them. This order may not correspond to the order in which the client sent the requests.

The general client discovers the current contents of the queue by doing a series of *checks* on the exchange. The first check tells the client about the oldest response in the queue; the second check tells about the second-oldest response, and so on. When the client sees a response it wants to read, it must *copy* it from the queue; this removes the response from the queue and makes subsequent checks start over with the oldest response still on the queue.

There are two ways to do checks. They differ only in their action when their are no new responses to report on:

- **ExCheck** checks the queue. If there are no new responses (responses that are not yet returned to a wait or check), it returns immediately with the value -1 and sets the external variable **errno** to **ENOMSG**.
- **ExWait** checks the queue. If there are no new responses (responses that are not yet returned to a wait or check), it waits until a new response arrives.

These are the definitions of **exCheck** and **exWait**:

```
exCheck(exch, mstat);
unsigned short exch;
struct msgret *mstat;

exWait(exch, mstat);
unsigned short exch;
struct msgret *mstat;
```

Exch is the exchange descriptor; **mstat** points to a structure to which the kernel writes a report on the message. **ExCheck** and **exWait** return -1 on error or failure.

This is the definition of the **msgret** structure in the include file.

```
struct msgret {
        unsigned short m_rqCode;
        unsigned short m_reqdest;
        int m_size;
        char m_flag;
        unsigned short m_ercRet;
        unsigned char m_cputype;
        unsigned char m_slot;
        struct rqheader *m_offset;
};
```

When **exCheck** or **exWait** report on a new response, the returned **msgret** structure looks like this:

m_rqCode The service that is responding.

m_reqdest The request descriptor. This is the same value the client got from **exRequest** when it sent the request.

m_size Size of the response. This is the size of the request block, plus the sizes of any response data blocks, plus the number of bytes used to make the response data blocks begin on even addresses.

m_flag The value MRESPONSE.

m_ercRet The server's error return code. This is same value that is in **r_ercRet** in the response header.

The remaining fields give information on the physical location of the message and its host CPU.

If a general client wants to copy the response, it must set up a request block to receive the data. This block has the same format and data as the one passed to **exRequest**. In addition, the client must provide

the address of each response data area; this goes in the **pc_count** fields of each response PbCb.

The response area sizes need not be the same sizes specified in the request: the client may want to specify bigger areas if the size of the response (**m_size** in the **msgret** structure) indicates that the server ignored the client's original response data maximums. In any case, the kernel right-truncates any response data blocks that will not fit the client's areas.

It is reasonable for a client to set up a single request block, with complete response PbCbs, and use this block both for sending the request and for copying the response.

Example

Figure 14–7 shows a request block ready to receive a response from **JEEVES**.

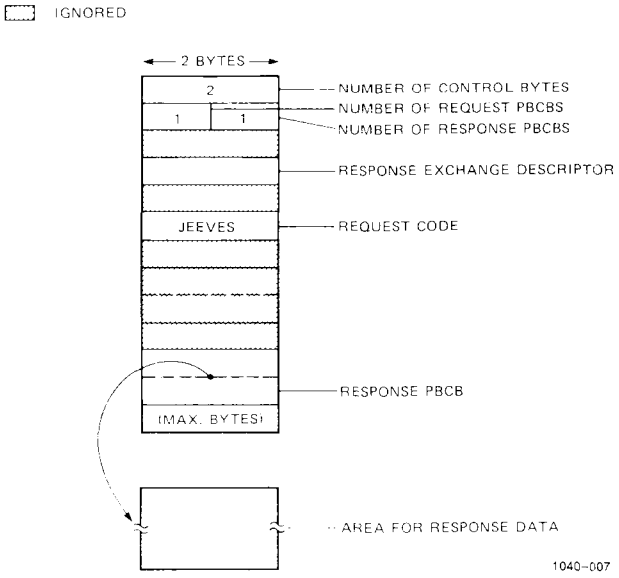


Figure 14-7. Ready for Response from **JEEVES**

Reqdes is the same request descriptor returned by **exRequest** and in **m_reqdes** in the response's **msgret** structure.

The **exCpResponse** system call copies a response:

```
exCpResponse(reqdes, reqblp)
unsigned short reqdes;
struct rqheader *reqblp;
```

Reqdes is the request descriptor; **reqblp** points to the request block that is to receive the reply. The kernel modifies,

- **r_ercRet** in the header to the server's return code;
- each data area indicated by a response PbCb.

ExCpResponse modifies the same request block fields as does **exCall**. See Figure 14-5

Mixing General and Basic Roles

A single client can use both **exCall** and **exRequest**. A response to an **exCall** request is handled separately from other responses going to the same exchange. **ExCall** does not affect the status of any message anywhere, except its own request and response.

The Server

The server behaves very much like the general client: it monitors queues of incoming messages (requests) and it has a single, independent, stream of outgoing messages (responses). To receive requests, the process must associate an exchange with a request code.

Choosing the Request Code

Convergent Technologies reserves all request codes less than 49152 (0xC000). Any service you add must have a request code between 49153 (0xC001) and 65535 (0xFFFF).

Choose new request codes in as few continuous sequences as you can. Each continuous sequence of request codes requires a certain amount

of overhead in certain system tables. It is very easy to fill up these tables.

Setting Up the Service

A server, like a general client, must provide one exchange for each message queue. It can use the standard exchange specified by **ExQueryDfltRespExch**, or it can allocate additional exchanges with **ExAllocExch**.

A server then assigns each of its exchanges one or more services. It does this with **exServeReq**:

```
exServeReq(exch, code)
unsigned short exch;
int code;
```

Exch is an exchange descriptor. **Code** is the request code for the service. The first process to specify a request code with **exServeReq** is the only server for that request code.

The Request Side

The server, like the general client, monitors its message queues with **exCheck** or **exWait**. **ExCheck** and **exWait** deal with request queues in precisely the way they deal with response queues. We must, however, take a second look at the **msgret** structure.

m_rqCode The request code specified by the client. This distinguishes services when a single exchange is associated with more than one service

m_reqdest The request descriptor. This uniquely identifies the request and its response in subsequent server system calls.

m_size Size of the request. This is the request block, plus any request data, plus any padding to make blocks of request data appear at even addresses.

m_flag The value **MREQUEST**.

m_ercRet No useful value.

The values in the other fields serve no purpose in the current implementation.

The server must dispose of each request on the queue. The **exCpRequest** call copies a request:

```
exCpRequest(reqdes, reqblp)
unsigned short reqdes;
struct rqheader *reqblp;
```

As with **exCpResponse**, **exCpRequest** removes the message from the queue and restarts at the oldest message subsequent checking on the queue.

ExCpRequest copies the message to the memory pointed to by **reqblp**; **reqblp** must be an even address. **ExCpRequest** puts the request data blocks directly after the request block; each data block begins at an even address. **ExCpRequest** modifies the request PbCbs so that they describe the new copies of the request data blocks. With this exception, the request block and data blocks all contain the values placed in them by the client. The response exchange descriptor and response data pointers, however, are meaningless to the server.

Example

The **JEEVES** server is run when the system first comes up. It begins by appropriating its request code:

```
ex = ExQueryDfltRespExch();
if (exServeReq(ex, JEEVES) == -1) {
        perror("JEEVES server:  can't get request code");
        exit(1);
}
```

The server must then periodically check its exchange. When it discovers a request it wants to deal with, the server copies the request:

```
jmess = (struct jblock *) malloc(mstat.m_size)
exCpRequest(mstat.m_reqdest, jmess);
```

Figure 14-8 shows a request received by the **JEEVES** server.

The Response Side

The server uses a request block to describe a response. It sets

- **R_ercRet** in the request block header. This indicates the return code.

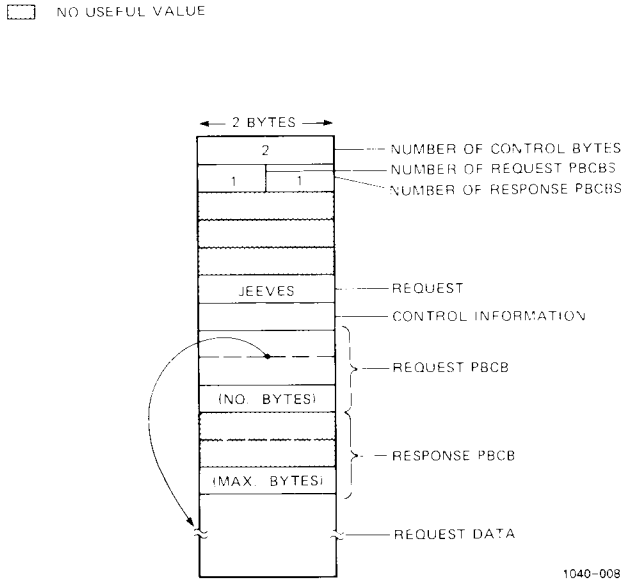


Figure 14-8. Request Copied by **JEEVES** Server

- Each response PbCb. Each PbCb must describe the location and size of a block of response data.

It is reasonable, but not necessary, for a server to use the same request block that it copied from its request queue.

The server sends the response with **exRespond**:

```
exRespond(reqdes, reqblp)
unsigned short reqdes;
struct rqheader *reqblp;
```

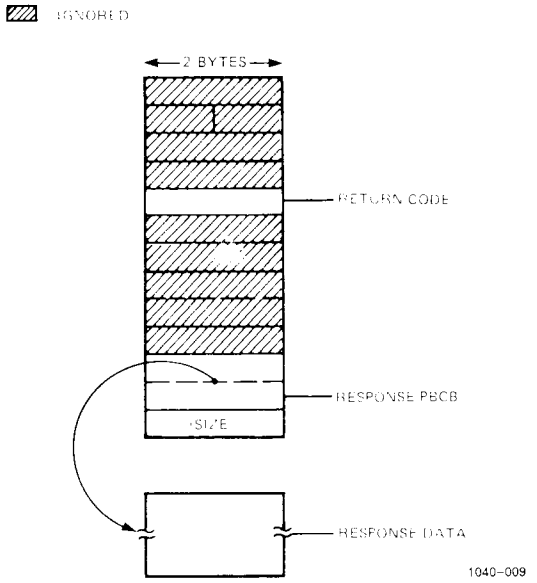


Figure 14-9. **JEEVES** Server Ready to Send Response

Reqdes is a request descriptor that indicates which request provoked the response. **Reqblp** points to the request block that describes the response.

Example

Figure 14–9 shows a response set up by the **JEEVES** server.

Mixing Roles

A process that provides one or more services (playing the role of server) may use other services (playing the role of client). It is all right for such a process to use a single exchange for both requests and responses. On such an exchange, the requests and responses form a single queue of messages.

The mixed-role process must apply **exCpRequest** and **exReject** only to requests and **exCpResponse** only to responses.

Note that **exCall** does not affect the status of any waiting request or response.

Example

```
exServeReq(exchange = ExQueryDfltRespExch(), JEEVES);
for (;;) {
        exWait(exchange, &mstat)
        switch (mstat.m_flag) {

        case MREQUEST:
                inblock = alloc(mstat.msize);
                exCpRequest(mstat.m_reqdes, inblock);
                .
                .
                (action on request, including exRespond)
                .
                .
                break;

        case MRESPONSE:
                exCpResponse(mstat.m_reqdes, outblock);
                .
                .
                (action on response)
                .
                .
                break;
}
```

Simplified Access to a Request Queue

Most servers will normally do an **exWait**, followed closely by an **exCpResponse**. These two actions are combined in **exGetNext**. It is different from a simple **exWait/exCpResponse** sequence in the following ways:

- It ignores responses.
- It is a single system call. This saves work by the kernel.
- Like **exCall**, it does not affect the status of any messages it does not copy.

ExGetNext takes the following form:

```
exGetNext(exch, reqblp, size)
unsigned short exch;
struct rqheader *reqblp;
unsigned short size
```

where

exch is the descriptor for the exchange of interest;

reqblp points to the buffer that is to receive the message;

size is the size of the buffer.

Any message that is too big for the buffer is right-truncated to fit.

Message Queue Access Summary

This section summarizes how a process accesses the messages in a message queue. Note that first-in first-out (FIFO) access is simplest, but not mandatory.

FIFO access to a message list occurs only when a process follows each check of the exchange (**exCheck** or **exWait**) with a copy (**exCpRequest**, **exCpResponse**) or disposal of the newly discovered message.

A process can thumb through the **msgret** structures for the entire message queue simply by not copying or discarding any message at the exchange between checks. Repeated calls to **exCheck** return successively newer **msgret** structures until there aren't any more; repeated calls to **exWait** return successively newer **msgret** structures forever.

Any copy or discard on a message waiting at an exchange resets the procedure for that exchange: subsequent checks start over with the oldest message.

ExCall, however, ignores all messages except the request it sends and the response it waits for and copies; **ExGetNext** ignores all messages except the request it copies.

One can think of each message as having a "noticed" flag. A successful **exCheck** or **exWait** always notices the oldest message whose "noticed" flag is off. The following events affect this flag:

- The message's arrival at the queue turns that message's "noticed" flag *off*.
- An **exCheck** or **exWait** that notices a message turns that message's "noticed" flag *on*.
- An **exCpResponse** or **exCpRequest** on a message in a queue turns off the "noticed" flag of every message in the queue.

ExCall and **exGetNext** do not affect the noticed flag of any message.

Releasing Unwanted Exchanges

All processes share a finite pool of exchanges. A process's death automatically deallocates its exchanges. A process that will continue working but has no further use for an exchange should relinquish it with **exDeallocExch**:

```
exDeallocExch(exch)
unsigned short exch;
```

Exch is the descriptor of the unwanted exchange.

The death of a process deallocates all its exchanges. Process death is the only way to deallocate the default response exchange.

The kernel has standard actions for messages still waiting at an exchange when the exchange is deallcoated. The kernel discards any responses still waiting at the deallocated exchange. The kernel issues an **exReject** on any requests still waiting at the deallocated exchange:

```
exReject(rd, ercServiceNotAvail);
```

where **rd** is the request's request descriptor, and **ercServiceNotAvail** is a constant defined in the include file.

Setting Final Action

The deallocation of a request's exchange may be a signal that the request is obsolete. For example, a client might terminate suddenly before its server has acted on a particularly time-consuming request. It is useful for a client to prepare, *in advance*, messages that indicate that it is no longer able to wait for a response. **ExSendOnDealloc** provides a way to tie a request to the deallocation of a particular exchange.

```
exSendOnDealloc(reqblp)
struct rqheader *reqblp;
```

This call specifies a *final request*. **Reqblp** points to the request block; **exSendOnDealloc** returns a request descriptor. **ExSendOnDealloc** causes the kernel to copy the final request indicated by **reqblp**; but the kernel does not immediately queue the request at the server's exchange. The status of the final request's response exchange controls the final request: when this exchange is deallocated, the kernel queues the final request. Thus the client dispatches the final request either by dying or by a call to **exDeallocExch**.

Note that a server must respond to a final request, even though the client will never read the response.

A client can cancel a final request:

```
exCnxSendOnDealloc(req)
unsigned short req;
```

Req is the request descriptor from the call to **exSendOnDealloc**.

Summary of System Calls

Here is a complete summary of ICC system calls. On error or failure, all these calls return –1 and set the external variable **errno**. Only those functions indicated return a useful value on success. For **errno** values, see *intro*(3) and *perror*(3) in the *MegaFrame CTIX Operating System Manual*. Always test for equality with -1: some calls return valid negative values. Only indicated functions return useful values on success.

Obtain and abandon exchanges.

exQueryDfltRespExch()
Returns the exchange descriptor for the default response exchange. Despite the name, you must specify this exchange explicitly, and you can accept requests on a default response exchange.

exAllocExch()
Allocates an exchange and returns the exchange descriptor.

exDeallocExch(ex)
unsigned short ex;
Deallocates the exchange specified by **ex**.

Call a server.

exCall(reqblk)
struct rqheader *reqblk;
Sends the request described by request block **reqblk**; waits for the response; copies the response. **ExCall** ignore all messages except its own request and response.

Send a message.

exRequest(reqblp)
struct rqheader *reqblp;
Transmits the request described by request block **reqbl**. Returns a request descriptor.

exRespond(reqdes, reqblp)
unsigned short reqdes;
struct rqheader *reqblp;
Transmits a response to the request indicated by **reqdes**. **Reqblp** describes the response.

Check for waiting messages.

exWait(ex, mstat);
unsigned short ex;
struct msgret *mstat;
Gets status structure for oldest unnoticed message waiting at exchange specified by **ex**. **ExWait** writes the status structure to the location pointed to by **mstat**. If there are no unnoticed messages at the specified exchange **exWait** waits until there are.

exCheck(ex, mstat);
unsigned short ex;
struct msgret *mstat;

Gets status structure for oldest unnoticed message waiting at exchange specified by **ex**. **ExCheck** writes the status structure to the location pointed to by **mstat**. If there are no messages at the specified exchange, **exCheck** sets **errno** to **ENOMSG** and returns -1.

A successful **exWait** or **exCheck** marks the discovered message as "noticed": subsequent checks on the same exchange produce only newer messages. Any disposal of a message waiting at an exchange (**exCpRequest**, **exCpResponse**, **exReject**, but not **exCall** or **exGetNext**) marks all messages on the exchange as "unnoticed": subsequent checks on the exchange start over with the very oldest message on the exchange.

Dispose of a response.

exCpResponse(reqdes, reqst)
short reqdes;
struct request *reqst;

Copies the response indicated by **reqdes**.

Dispose of a request.

exCpRequest(reqdes, reqst)
unsigned short reqdes;
struct request *reqst;

Does not copy the *request* indicated by **reqdes**; sets the request's return code field to **code** and sends the message back as a response. **ExReject** does not permit the server to examine the control data or other request fields.

Get the next request.

exGetNext(exch, reqblp, size)
unsigned short exch;
struct rqheader *reqblp;
unsigned short size

Copies the oldest request waiting at **exch** to **reqblp**. If the request is more than **size** bytes, it is right-truncated.

Set final action.

short exSendOnDealloc(reqblk)
struct rqheader *reqblk;

Reads the request described by the request block pointed to by **reqblk**, but doesn't send it until this process dies or otherwise deallocates the request's response exchange. Returns a request descriptor that is only good for cancelling the request.

A Note on Communicating with CTOS

Integer value conversion is necessary if client and server don't both run under CTIX. CTIX runs on Application Processors, which are based on the Motorola 68010 and 68020. CTOS runs on all others S/1280 processors, which are based on the INTEL 80186. Motorola and INTEL use opposite byte ordering for integer values. ICC does this conversion automatically for the request block header. For integers in the control information and in the request and response data, the programmer must do the conversion. See *swapshort*(3C) in the *MegaFrame CTIX Operating System Manual.*

CTIX ICC for the CTOS Programmer

ICC differs in important ways from its CTOS counterpart, Inter-Process Communication (IPC). For the benefit of programmers whose previous experience is with CTOS, we summarize these differences.

System Primitives

Most of the differences between ICC primitives and IPC primitives reflect an important difference between CTIX and CTOS: CTIX processes cannot access each other's memory. Thus CTIX processes must send and receive copies of messages, not just pointers to messages.

ExCall provides services not available under CTOS. Its use is strongly encouraged because it requires less work by the CTIX kernel.

CTIX does not support an equivalent of **Send** or **Psend**.

Exchanges and Messages

A CTIX exchange can serve as both request and response exchange.

A CTIX exchange identifier is called an *exchange descriptor*. Unlike CTOS identifiers, descriptors are not unique, except among exchanges owned by the same process.

Each CTIX exchange belongs to a specific process. Only the exchange's owner can receive messages at that exchange.

The CTIX programmer is encouraged to use explicitly the default response exchange. Use of this exchange in asynchronous messages is perfectly safe.

CTIX does not strictly enforce first-in-first-out access to an exchange's messages. **ExCall** sends a request and then waits for the corresponding response, ignoring all other responses. A certain usage of **exWait** and **exCheck** (see "Message Queue Access Summary", above) gives a process random access to an exchange's queue of messages.

15

CTIX System Assembler User's Guide

Introduction

This chapter is a CTIX system assembler user's guide for the Motorola 68010, 68020, and 68881. This chapter should be used in conjunction with the *MC68000 16 Bit Microprocessor User's Manual**, the *MC68020 32-Bit Microprocessor User's Manual***, and the *MC68881 Floating-Point Coprocessor User's Manual*†. Programmers familiar with the MC68010, MC68020, and MC68881 should be able to program in CTIX assembly language by referring to this chapter; however, this is not a manual for a processor itself. Details about the effects of instructions, meanings of status register bits, handling of interrupts, and many other issues are not dealt with here.

NOTE

This chapter refers to the MC68010, MC68020, and MC68881 as "MC68K" processors. This chapter also refers to the MC68000 *16-Bit Microprocessor User's Manual*, the *MC68020 32-Bit Microprocessor User's Manual*, and the *MC68881 Floating-Point Coprocessor User's Manual* as "MC68K user manuals."

* *MC68000 16-BIT MICROPROCESSOR User's Manual*, Third Edition; Englewood Cliffs, N.J.: Prentice-Hall, 1982.

** *MC68020 32-BIT MICROPROCESSOR User's Manual*, Second Edition; Englewood Cliffs, N.J.: Prentice-Hall, 1985.

† *MC68881 FLOATING-POINT COPROCESSOR User's Manual*, First Edition, 1985.

Organization

This section describes

- how to invoke the assembler
- general syntax rules and program organization
- pseudo-operations, span-dependent optimization, and address modes
- major differences between CTIX and other assemblers

Table 15-3 presents all MC68K instruction formats alphabetically by operation.

Use of the Assembler

The CTIX system command **as** invokes the assembler and has the following syntax:

as [**–o** *output*] *file*

This causes the named *file* to be assembled. The output of the assembly is left on the file *output* specified by the **-o** flag. If no such specification is made, the output is left in a file whose name is formed by removing the **.s** suffix, if there is one, from the input file name and appending an **.o** suffix.

The CTIX assembler, versions 5.0 and higher, supports flexnames. "Flexname" is the AT&T term for the feature that allows a program identifier to be longer than eight characters. The following are available as backward compatibility options: the **–T** option to **as**, which truncates symbols to eight characters, and the **–G** option to **ld**, which changes the symbol name look-up algorithm. (Refer to *as*(1) and *ld*(1) in the *CTIX Operating System Manual*).

General Syntax Rules

Assembler Code Elements

Identifiers, registers, constants, and comments are basic elements that typically occur in CTIX assembly code. The following paragraphs describe each of these elements.

Identifiers

An identifier is a string of characters taken from the set **a–z**, **A–Z**, **_**, **~**, **%**, and **0–9**. The first character of an identifier must be a letter (upper or lower case) or an underscore. Upper and lower case letters are distinguished. For example,

con35 and CON35

are two distinct identifiers.

There is no limit on the length of an identifier.

The value of an identifier is established by the set pseudo-operation or by using it as a label. (For more information on set pseudo-operations and labels, see "Symbol Definition Operations" and "Location Counters and Labels," respectively, below.)

The tilde (~) has special significance to the assembler. A ~ used alone as an identifier means "the current location." A ~ used as the first character in an identifier becomes a "." in the symbol table. This allows symbols such as **.eos** and **.Ofake** to make it into the symbol table (as required by the Common Object File Format.)

Registers

An MC68K processor register is an identifier preceded by the character **%**. The predefined registers are

%d0	%d4	%a0	%a4	%f0	%f4	%cc	%usp	%isp
%d1	%d5	%a1	%a5	%f1	%f5	%pc	%caar	%msp
%d2	%d6	%a2	%a6	%f2	%f6	%sp	%cacr	%sfc
%d3	%d7	%a3	%a7	%f3	%f7	%sr	%dfc	%vbr
								%fp

NOTE

The registers **%a7** and **%sp** represent the same machine register. Likewise, **%a6** and **%fp** are equivalent.

Constants

The assembler deals with integer and floating-point constants. They may be entered as decimal, octal, hexadecimal, or character constants. Internally, the assembler treats all constants as 32-bit binary two's complement quantities or as floating point, where applicable.

Numerical Constants. All numerical constants must be preceded by ampersand (**&**). Otherwise, the number is assumed to be an absolute address.

A decimal constant is a string of digits beginning with a non-zero digit.

An octal constant is a string of digits beginning with zero.

A hexadecimal constant consists of the characters **0x** or **0X** followed by a string of characters from the set **0–9**, **a–f**, and **A–F**. In hexadecimal constants, upper and lower case letters are not distinguished.

Floating point constants are preceded by **0f**. Double floating point constants are preceded by **0d**.

Examples of decimal, octal, hexadecimal, and floating point constants are listed as follows:

```
set     const,&35       # Decimal 35
mov.w   &035,%d1        # Octal 35 (decimal 29)
set     const,&0x35     # Hex 35 (decimal 53)
mov.w   &0xff,%d1       # Hex ff (decimal 255)
fmov.l  &0f123,%f0      # Floating point 123
```

Character Constants. All character constants must be preceded by ampersand (**&**). Character constants may be ordinary or special.

An *ordinary character constant* consists of a single-quote (') followed by an arbitrary ASCII character other than \. The value of the constant is equal to the ASCII code for the character. Special meanings of characters are overridden when used in character constants. For example, if '# is used, the # is not treated as introducing a comment.

A *special character constant* consists of '\ followed by another character. All of the special character constants, and examples of ordinary character constants, are listed as follows:

Constant	*Value*	*Meaning*
&'\b	0x08	Backspace
&'\t	0x09	Horizontal Tab
&'\n	0x0a	Newline (Line Feed)
&'\v	0x0b	Vertical Tab
&'\f	0x0c	Form Feed
&'\r	0x0d	Carriage Return
&'\\	0x5c	Backslash (\)
&''	0x27	Single Quote
&'0	0x30	Zero
&'A	0x41	Capital A
&'a	0x61	Lowercase A

Comments

Comments are introduced by the character # and continue to the end of the line. Comments may appear anywhere and are ignored by the assembler. (See also the discussion of the **ident** pseudo-operation in "Location Counter Control and Other Section Operations" below.)

Format of Assembly Language Line

Typical lines of CTIX assembly code look like:

```
# Clear a block of memory at location %a3

        text    2
        mov.w   &const,%d1
loop:   clr.l   (%a3)+
        dbf     %d1,loop        # go back for const
                                # repetitions
init2:
        clr.l count; clr.l credit; clr.l debit;
```

These general points about the example should be noted:

- An identifier occurring at the beginning of a line and followed by a colon (:) is a label. One or more labels may precede any assembly language instruction or pseudo-operation. Refer to "Location Counters and Labels," below, for more information.
- A line of assembly code need not include an instruction. It may consist of a comment alone (introduced by #), a label alone (terminated by :), or it may be entirely blank.
- It is good practice to use tabs to align assembly language operations and their operands into columns, but this is **not** a requirement of the assembler. An opcode may appear at the beginning of the line, if desired, and spaces may precede a label. A single blank or tab suffices to separate an opcode from its operands. Additional blanks and tabs are ignored by the assembler.
- It is permissible to write **several** instructions on one line by separating them by semicolons. The semicolon is syntactically equivalent to a newline. A semicolon inside a comment is ignored.

Program Organization

A CTIX program is organized in section that are maintained and identified by location counters and labels, respectively. The following paragraphs describe sections, location counters, and labels.

Sections

A program in CTIX assembly language is normally broken into text, data, or bss sections. Instructions are usually placed in text sections, initialized data in data sections, and uninitialized data in bss sections. However, the assembler does not enforce this convention; for example, it permits intermixing of instructions and data in a text section.

The assembler permits up to nine separate sections, three of which must be text, data, and bss. (User-defined sections and comment sections are discussed below in "Location Counter Control and Other Section Operations.") The assembly language program may switch freely between them by using assembler pseudo-operations. (See "Location Counter Control and Other Section Operations," below.) When generating the object file, the assembler concatenates all sections of the same name to generate a single section of that name. For example, if there are no section or ident pseudo-operations in the source file, the object file contains only one text section and one data section.

There is only one bss section to begin with, and it maps directly into the object file.

Because the assembler keeps together everything from a given section when generating the object file, the order in which information appears in the object file may not be the same as in the assembly language file. For example, if the data for a program consists of

```
data    1           # section 1
short   0x1111
data    0           # section 0
long    0xffffffff
data    1           # section 1
byte    0x2222
```

then equivalent object code would be generated by

```
data0
long0xffffffff
short0x1111
byte0x2222
```

Location Counters and Labels

The assembler maintains separate location counters for each section. The location counter for a given section is incremented by one for each byte generated in that section.

The location counters allow values to be assigned to labels. When an identifier is used as a label in the assembly language input, the current value of the current location counter is assigned to the identifier. The assembler also keeps track of which section the label appeared in. Thus, the identifier represents a memory location relative to the beginning of a particular section. See "Location Counter Control and Other Section Operations," below, for more information on values that are assigned to location counters.

Types

Identifiers and expressions may have values that are absolute, relocatable, or undefined external types.

In the simplest case, an expression (or identifier) may have an *absolute value*, such as 29, -5000, or 262143.

An expression (or identifier) may have a value relative to the start of a particular section. Such a value is known as a *relocatable value*. The memory location represented by such an expression cannot be known at assembly time, but the relative values (i.e., the difference) of two such expressions can be known if they refer to the same section. Identifiers that appear as labels have relocatable values.

If an identifier is never assigned a value, it is assumed to be an *undefined external*. Such identifiers may be used with the expectation that their values will be defined in another program, and hence known at load time; but the relative values of undefined externals cannot be known.

Expressions

All constants are absolute expressions. An identifier may be thought of as an expression having the identifier's type. For conciseness, the following abbreviations are used:

abs	absolute expression
rel	relocatable expression
ext	undefined external

Expressions, except for floating point, may be built up from lesser expressions using the following operators:

Add	+
Subtract	–
Multiply	*
Divide	/
Left shift	<<
Right shift	>>
Exclusive-OR	^
Bit	$
One's complement	!
Modulus	%%
OR	|
AND	&&

Parentheses may be used to coerce the order of evaluation; otherwise the operators are evaluated, from highest to lowest precedence, as follows:

unary minus, !	Highest
*, /, %%	
+, binary minus	
<<, >>	
&&, |, ^	Lowest

Division by zero is illegal.

Expressions may be built up from lesser expressions using +, –, *, and / according to the following type rules:

abs + abs = abs
abs + rel = rel + abs = rel
abs + ext = ext + abs = ext

abs – abs = abs
rel – abs = rel
ext – abs = ext
rel – rel = abs, if the two relocatable expressions
are relative to the same section

abs * abs = abs
abs / abs = abs
–abs = abs

Expressions using <<, >>, ^, **$**, **!**, **%%**, **|**, and **&&** only make sense in absolute expressions.

CAUTION

Use of a **rel-rel** expression is dangerous, particularly when dealing with identifiers from text sections. The problem is that the assembler will determine the value of the expression before it has resolved all questions about span-dependent optimizations. Use this feature at your own risk!

Pseudo-Operations

Pseudo-operations establish values for identifiers and expressions. The following paragraphs identify data initialization, symbol definition, location counter control, symbolic debugging, and switch table pseudo-operations.

Data Initialization Operations

Data initialization operations establish values for expressions. Each operation is presented by operation name, followed by the argument types and a description of the operation.

byte abs,abs,...

One or more arguments, separated by commas, may be given. The values of the arguments are computed to produce successive bytes in the assembly output.

short abs,abs,...

One or more arguments, separated by commas, may be given. The values of the arguments are computed to produce successive 16-bit words in the assembly output.

long expr,expr,...

One or more arguments, separated by commas, may be given. Each expression may be absolute, relocatable, or undefined external. A 32-bit quantity is generated for each such argument (for relocatable or undefined external expressions, the value may not be filled in until load time).

ASCII string,string,...

One or more strings may be given, separated by commas. The values of the arguments are computed to produce ASCII strings.

float fexpr,fexpr,...

One or more arguments, separated by commas, may be given. The values of the arguments are computed to produce 32-bit floating point values. *fexpr* is a floating point constant.

double fexpr,fexpr,...

One or more arguments, separated by commas, may be given. The values of the arguments are computed to produce 64-bit floating point values. *fexpr* is a floating point constant.

space abs

The value of *abs* is computed, and the resultant number of bytes of zero data is generated. For example,

```
space    6
```

is equivalent to

```
byte     0, 0, 0, 0, 0, 0
```

Alternatively, arguments may be bit-field expressions. A bit-field expression has the form

```
n:value
```

where both *n* and *value* denote absolute expressions. The quantity *n* represents a field width; the low-order *n* bits of *value* become the contents of the bit-field. Successive bit-fields fill up 32-bit long quantities starting with the high-order part. If the sum of the lengths of the bit-fields is less than 32 bits, the assembler creates a 32-bit long with zeros filling out the low-order bits. For example,

```
long4:-1, 16:0x7f, 12:0, 5000
```

and

```
long4:-1, 16:0x7f, 5000
```

are equivalent to

```
long0xf007f000, 5000
```

Bit fields may not span pairs of 32-bit longs. Thus

```
long24:0xa, 24:0xb, 24:0xc
```

yields the same thing as

```
long0x00000a00, 0x00000b00, 0x00000c00
```

Symbol Definition Operations

Symbol definition operations establish values for identifiers. Each operation is presented by operation name, followed by the argument types and a description of the operation.

set identifier,expr

The value of *identifier* is set equal to *expr*, which may be absolute or relocatable.

comm identifier,abs

The named identifier is to be assigned to a common area of size *abs* bytes. If *identifier* is not defined by another program, the loader will allocate space for it.

The type of *identifier* becomes undefined external.

lcomm identifier,abs

The named identifier is assigned to a local common of size *abs* bytes. This results in allocation of space in the bss section.

The type of *identifier* becomes relocatable.

global identifier

This causes *identifier* to be externally visible. If *identifier* is defined in the current program, then declaring it global allows the loader to resolve references to *identifier* in other programs.

If *identifier* is not defined in the current program, the assembler expects an external resolution; in this case, therefore, *identifier* is global by default.

Location Counter Control and Other Section Operations

Location counter control operations establish values for location counters. Each operation is presented by operation name, followed by the argument types and the operation description.

data abs

The argument, if present, must evaluate to 0, 1, 2, or 3; this shows the number of the data section into which assembly is to be directed. If no argument is present, assembly is directed into data section 0. (Note that the *abs* argument may not be supported in future releases.)

text abs

The argument, if present, must evaluate to 0, 1, 2, or 3; this shows the number of the text section into which assembly is to be directed. If no argument is present, assembly is directed into text section 0.

Before the first data or text operation is encountered, assembly is, by default, directed into text section 0. (Note that the *abs* argument may not be supported in future releases.)

ident "comment string"

This operation causes a comment section to be created if there is no comment section already and puts the *comment string* into the comment section. This is used for SCCS "what" strings [see *what*(1) in the *CTIX Operating System Manual*].

(Not available on versions of the CTIX assembler previous to 6.0.)

section name,"flag"

This operation causes the creation of a new section, which is identified by *name* and has an attribute corresponding to *flag*. Attributes are defined in <*sys/scnhdr.h*>. The argument *flag* and associated attribute is one of the following:

b	STYP_BSS
c	STYP_COPY
i	STYP_INFO
d	STYP_DSECT
x	STYP_TEXT
n	STYP_NOLOAD
o	STYP_OVER
l	STYP_LIB
w	STYP_DATA

NOTE: A new section is created as described above, but it is not loaded by the CTIX kernel unless there is an *ifile* for it and the *ifile* has been linked. (An *ifile* is an **ld** input file containing link editor command language scripts.) For information on creating an *ifile*, see "The Link Editor" in *Programmer's Guide: UNIX System V*.

(Not available on versions of the CTIX assembler previous to 6.0.)

org expr

The current location counter is set to *expr*. *Expr* must represent a value in the current section, and must not be less than the current location counter.

even

The current location counter is rounded up to the next even value.

align abs

The argument must evaluate to 0, 1, 2, 3, or 4; this shows the alignment of the following instructions or data to the nearest *abs* byte boundary. For example, *even* is the same as *align 2*.

Symbolic Debugging Operations

The special assembler pseudo-operations, **file** and **ln**, and the symbol attribute operations place debugging information into the object code file. This information typically includes line numbers and information about C language symbols, such as their type and storage class. The CTIX system C compiler generates symbolic debugging information when the **-g** option is used. Assembler programmers may also include such information in source files.

file and ln

The **file** pseudo-operation passes the name of the source file into the object file symbol table. It has the form

```
file "filename"
```

where *filename* consists of one to 14 characters.

The **ln** pseudo-operation makes a line number table entry in the object file. That is, it associates a line number with a memory location. Usually the memory location is the current location in text. The format is

```
ln line [,value]
```

where *line* is the line number. The optional *value* is the address in the text, data, or bss section that is associated with the line number. The default when *value* is omitted (which is usually the case) is the current location in the text section.

Symbol Attribute Operations

The symbol attribute operations include basic symbolic testing and attribute-assigning pseudo-operations.

Basic Symbolic Testing Operations. The basic symbolic testing pseudo-operations are **def** and **endef**. These operations enclose other pseudo-operations that assign attributes to a symbol and must be paired, as follows:

```
def         name
.                       # Attribute
.                       # Assigning
.                       # Operations
endef
```

As an example,

```
defx
val-16
scl1
type06
endef
```

is equivalent to the C language statement

```
int x;
```

NOTE

def does not define the symbol, although it does create a symbol table entry. Because an undefined symbol is treated as external, a symbol that appears in a **def** but never acquires a value will eventually result in an error at link edit time. To allow the assembler to calculate the sizes of functions for other CTIX system tools, each **def/endef** pair that defines a function name must be matched by a **def/endef** pair after the function in which a storage class of **-1** is assigned.

Attribute-Assigning Operations. The attribute-assigning operations described below apply to the symbol *name* that appears in the opening **def** pseudo-operation in "Basic Symbolic Testing Operations," above.

val expr

Assigns the value *expr* to *name*. The type of the expression *expr* determines with which section name is associated. If value is ~, the current location in the text section is used.

scl expr

Declares a storage class for *name*. The expression *expr* must yield an ABSOLUTE value that corresponds to the C compiler's internal representation of a storage class. The special value **-1** designates the physical end of a function.

type expr

Declares the C language type of *name*. The expression *expr* must yield an ABSOLUTE value that corresponds to the C compiler's internal representation of a basic or derived type.

tag str

Associates *name* with the structure, enumeration, or union named *str* that must have already been declared with a **def/endef** pair.

line expr

Provides the line number of *name*, where *name* is a block symbol. The expression *expr* should yield an ABSOLUTE value that represents a line number.

size expr

Gives a size for *name*. The expression *expr* must yield an ABSOLUTE value. When *name* is a structure or an array with a predetermined extent, *expr* gives the size in bytes. For bit fields, the size is in bits.

dim expr1, expr2,...

Indicates that *name* is an array. Each of the expressions (expr1, expr2,...) must yield an ABSOLUTE value that provides the corresponding array dimension.

Switch Table Operation

In the switch table operation, the CTIX system C compiler generates a compact set of instructions, determined by a special **swbeg** operation, for the C language switch construct. For example,

```
sub.l     &1,%d0
cmp.l     %d0,&4
bhi       L%21
add.w     %d0,%d0
mov.w     10(%pc,%d0.w),%d0
jmp       6(%pc,%d0.w)
swbeg     &5

L%22:
        shortL%15-L%22
        shortL%21-L%22
        shortL%16-L%22
        shortL%21-L%22
        shortL%17-L%22
```

The special **swbeg** pseudo-operation communicates to the assembler that the lines following it contain rel-rel subtractions. Remember that ordinarily such subtractions are risky because of span-dependent optimization. Here, however, the assembler makes special allowances for the subtraction because the compiler guarantees that both symbols will be defined in the current assembler file, and that one of the symbols is a fixed distance away from the current location.

The **swbeg** pseudo-operation takes an argument that looks like an immediate operand. The argument is the number of lines that follow **swbeg** and that contain switch table entries. **Swbeg** inserts two words into text. The first is the ILLEGAL instruction code. The second is the number of table entries that follow. The CTIX system disassembler needs the ILLEGAL instruction as a hint that what follows is a switch table. Otherwise it would get confused when it tried to decode the table entries (differences between two symbols) as instructions.

Span-Dependent Optimization

The assembler makes certain choices about the object code it generates based on the distance between an instruction and its operand(s). Choosing the smallest, fastest form is called span-dependent optimization. Span-dependent optimization occurs most obviously in the choice of object code for branches and jumps. It also occurs when an operand may be represented by the program counter relative address mode instead of an absolute two-word (long) address.

The span-dependent optimization capability is normally enabled; the **–n** command line flag disables it. When this capability is disabled, the assembler makes worst-case assumptions about the types of object code that must be generated.

The CTIX system compiler generates branch instructions without a specific offset size. When the optimizer is used, it identifies branches that could be represented by the short form, and it changes the operation accordingly. The assembler only chooses between long and very-long representations for branches.

Branch instructions, such as **bra**, **bsr**, and **bgt**, can have either a byte or a word pc-relative address operand. A byte size specification should be used only when the user is sure that the address intended can be represented in the byte allowed. The assembler will take one of these instructions with a byte size specification and generate the byte form of the instruction without asking questions.

Although the largest offset specification allowed is a word, large programs could conceivably have need for a branch to a location not reachable by a word displacement. Therefore, equivalent long forms of these instructions might be needed. When the assembler encounters a branch instruction without a size specification, or with a word size specification, it tries to choose between the long and very long forms of the instruction. If the operand can be represented in a word, then the word form of the instruction will be generated. Otherwise the very long form will be generated. For unconditional branches, e.g., **br**, **bra**, and **bsr**, the very long form is just the equivalent jump (**jmp** and **jsr**) with an absolute address operand (instead of pc-relative). For conditional branches, the equivalent very long form is a conditional branch around a jump, where the conditional test has been reversed.

Table 15–1 summarizes span-dependent optimizations. The assembler chooses only between the long form and very long form, while the optimizer chooses between the short and long forms for branches (but

not **bsr**).

TABLE 15-1

ASSEMBLER SPAN-DEPENDENT OPTIMIZATIONS

Instruction	Short Form	Long Form	Very Long Form
br, bra, bsr	byte offset	word offset	jmp or jsr with absolute long address
conditional branch	byte offset	word offset	short conditional branch with reversed condition around jmp with absolute long address
jmp, jsr		pc-relative address	absolute long address
lea.l, pea.l		pc-relative address	absolute long address

Address Mode Syntax

Table 15-2 summarizes the assembler syntax for the MC68K addressing modes. The following notations are used in this table:

n represents any digit from 0 to 7.

Rn, xn
Xn represent MC68K data or address registers.

d represents a displacement, and may stand for any absolute expression.

bd stands for an absolute expression that represents a base displacement.

od stands for an absolute expression that represents an optional outer displacement.

S (as in **%xn.S**) stands for one of the operation size attribute letters **b**, **w**, or **l**, representing a byte, word, or

long operation.

scale (as in **%xn.S*scale**) represents a value of 2, 4, or 8.

It is important to note that expressions used for the absolute addressing modes need not be absolute expressions in the sense defined in "Types," above. Although the addresses used in those addressing modes must eventually be filled in with constants, that can be done by the loader—there is no need for the assembler to compute them. The Absolute Long addressing mode is commonly used for accessing undefined external addresses.

TABLE 15-2

EFFECTIVE ADDRESS MODES (Page 1 of 2)

Motorola Syntax	CTIX Assembler Syntax	Effective Address Mode
Dn	%dn	Data Register Direct
An	%an	Address Register Direct
(An)	(%an)	Address Register Indirect
(An)+	(%an)+	Address Register Indirect with Postincrement
-(An)	-(%an)	Address Register Indirect with Predecrement
d_{16}(AN) **(d_{16},AN)**	d(%an)	Address Register Indirect with Displacement (d signifies a signed 16-bit absolute displacement)
d_8(An,Rn.W) d_8(An,Rn.L)	d(%an,%xn.w) d(%an,%xn.l)	Address Register Indirect with Index (d signifies a signed 8-bit absolute displacement)
(d_8,An,Xn.SIZE*SCALE)	**d(%an,%xn.S)**	**Address Register Indirect with Index (8-Bit Displacement)**
(bd,An,Xn.SIZE*SCALE)	**bd(%an,%xn.S)**	**Address Register Indirect with Index (Base Displacement)**
([bd,An],Xn.SIZE*SCALE,od)	**od([bd,%an],%xn.S*scale)**	**Memory Indirect Post-Indexed**
([bd,An,Xn.SIZE*SCALE]od)	**od([bd,%an,%xn.S*scale])**	**Memory Indirect Pre-Indexed**

Address modes in **boldface** are for the MC68020 processor.

TABLE 15-2

EFFECTIVE ADDRESS MODES (Page 2 of 2)

Motorola Notation	Assembler Notation	Effective Address Mode
xxx.W	xxx	Absolute Short Address (xxx signifies an expression yielding a 16- or 32-bit memory address)
xxx.L	xxx	Absolute Long Address (xxx signifies an expression yielding a 32-bit memory address)
LABEL(PC)	d(%pc)	Program Counter with Displacement (d signifies a signed 16-bit absolute displacement)
LABEL(PC,Rn.W) LABEL(PC,Rn.L)	d(%pc,%rn.w) d(%pc,%rn.l)	Program Counter with Index (d signifies a signed 8-bit absolute displacement)
(d_8,PC,Xn. SIZE*SCALE)	**d(%pc,%xn.S)**	**Program Counter Indirect with Index (8-Bit Displacement)**
(bd,PC, Xn. SIZE*SCALE)	**bd(%pc,%xn.S)**	**Program Counter Indirect with Index (Base Displacement)**
([bd,PC],Xn. SIZE.SCALE,od)	**od([bd,%pc], %xn.S*scale)**	**Program Counter Memory Indirect Post-Indexed**
([bd,PC,Xn. SIZE*SCALE], od)	**od([bd,%pc, %xn.S*scale**	**Program Counter Memory Indrect Pre-Indexed**
#xxx	&xxx	Immediate Data (xxx signifies an absolute constant expression)

Address modes in **boldface** are for the MC68020 processor.

Machine Instructions

Table 15-3 shows how MC68K instructions should be written for the assembler. The following notations are used in this table:

A (as in **add.A**) stands for one of the address operation size attribute letters **w** or **l**, representing a word or long operation.

S (as in **add.S**) stands for one of the operation size attribute letters **b**, **w**, or **l**, representing a byte, word, or long operation.

FMT (as in **fadd.FMT**) stands for

- one of the operation size attribute letters **b**, **w**, or **l**, representing a word or long operation.
- one of the floating-point data format attribute letters **s**, **d**, **x**, or **p**, representing a single precision, double precision, extended precision, or packed BCD operation.

CC In the contexts **bCC**, **dbCC**, and **sCC**, the letters **CC** represent any of the following condition code designations (except that **f** and **t** may not be used in the **bCC** instruction):

cc	carry clear
cs	carry set
eq	equal
f	false
ge	greater or equal
gt	greater than
hi	high
hs	high or same (=cc)
le	less or equal
lo	low (=cs)
ls	low or same
lt	less than
mi	minus
ne	not equal
pl	plus
t	true
vc	overflow clear
vs	overflow set

EA Represents an arbitrary effective address.

I An absolute expression, used as an immediate operand.

Q Represents an absolute expression evaluating to a number from 1 to 8.

L Represents a label reference, or any expression representing a memory address in the current section.

The following represent registers:

%ax	%dx	%fm
%ay	%dy	%fn
%an	%dn	

Refer to the notes under "meaning" in Table 15-3 for additional information concerning registers not listed here. Refer to Appendix C of the MC68020 user manual for instruction formats and the hex values of operation codes.

NOTE

MC68020 instruction formats are in **boldface**. MC68881 instruction formats begin with the letter **f**.

TABLE 15-3

MC68K INSTRUCTION FORMATS (Page 1 of 17)

Operation	Assembler Syntax		Meaning
ABCD	abcd.b	%dy,%dx -(%ay),-(%ax)	Add Decimal with Extend
ADD	add.S	EA,%dn %dn,EA	Add Binary
ADDA	add.A	Ea,%an	Add Address
ADDI	add.S	&I,EA	Add Immediate
ADDQ	add.S	&Q,EA	Add Quick
ADDX	addx.S	%dy,%dx -(%ay),-(%ax)	Add Extended
AND	and.S	EA,%dn %dn,EA	AND Logical
ANDI	and.S	&I,EA	AND Immediate
ANDI to CCR	and.b	&I,%cc	AND Immediate to Condition Codes
ANDI to SR	and.w	&I,%sr	AND Immediate to the Status Register (Privileged Instruction)
ASL	asl.S	%dx,%dy &Q,%dy	Arithmetic Shift (Left)
	asl.w	&I,EA	
ASR	asr.S	%dx,%dy &Q,%dy	Arithmetic Shift (Right)
	asr.w	&I,EA	
Bcc	bCC	L	Branch Conditionally (16-bit Displacement)

MC68020 instruction formats are in **boldface**.

MC68881 instruction formats begin with the letter **f**.

TABLE 15-3

MC68K INSTRUCTION FORMATS (Page 2 of 17)

Operation	Assembler Syntax		Meaning
	bCC.b	L	Branch Conditionally (Short) (8-bit Displacement)
BCHG	bchg	%dn,EA &I,EA	Test a Bit and Change Note: bchg should be written with no suffix. If the second operand is a data register, .l is assumed; otherwise .b is.
BCLR	bclr	%dn,EA &I,EA	Test a Bit and Clear Note: bclr should be written with no suffix. If the second operand is a data register, .l is assumed; otherwise .b is.
BFCHG	**bfchg**	**EA**	**Test Bit Field and Change**
BFCLR	**bfclr**	**EA**	**Test Bit Field and Clear**
BFEXTS	**bfexts**	**EA,%dn**	**Extract Bit Field Signed**
BFEXTU	**bfextu**	**EA,%dn**	**Extract Bit Field Unsigned**
BFFFO	**bfffo**	**EA,%dn**	**Find First One in Bit Field**
BFINS	**bfins**	**%dn,EA**	**Insert Bit Field**
BFSET	**bfset**	**EA**	**Set Bit Field**

MC68020 instruction formats are in **boldface**.
MC68881 instruction formats begin with the letter **f**.

TABLE 15-3

MC68K INSTRUCTION FORMATS (Page 3 of 17)

Operation	Assembler Syntax		Meaning
BFTST	**bftst**	**EA**	**Test Bit Field**
BKPT	**bkpt**	**&I**	**Breakpoint**
BRA	bra	L	Branch Always (16-bit Displacement)
	bra.b	L	Branch Always (Short) (8-bit Displacement)
	br br.b	L L	Same as bra Same as bra.b
BSET	bset	%dn,EA &I,EA	Test a Bit and Set Note: bset should be written with no suffix. If the second operand is a data register, .l is assumed; otherwise .b is.
BSR	bsr	L	Branch to Subroutine (16-bit Displacement)
	bsr.b	L	Branch to Subroutine (Short) (8-bit Displacement)
BTST	btst	%dn,EA &I,EA	Test a Bit and Set Note: btst should be written with no suffix. If the second operand is a data register, .l is assumed; otherwise .b is.
CALLM	**callm**	**&I,EA**	**CALL Module**

MC68020 instruction formats are in **boldface**.
MC68881 instruction formats begin with the letter **f**.

TABLE 15-3

MC68K INSTRUCTION FORMATS (Page 4 of 17)

Operation	Assembler Syntax		Meaning
CAS	**cas.S**	**%dc,%du,EA**	**Compare and Swap with Operand** **Note: c represents compare; u represents update value. See MC68020 User's Manual for details.**
CAS2	**cas2.A**	**%dc1:%dc2, %du1:%du2, (%rn1:%rn2)**	**Compare and Swap with Operand** **Note: c1 and c2 represent compare operands; u1 and u2 represent update values; rn1 and rn2 represent effective addresses. See MC68020 User's Manual for details.**
CHK	chk.w	EA,%dn	Check Register Against Bounds
CHK2	**chk2.S**	**EA,%rn**	**Check Register Against Bounds**
CLR	clr.S	EA	Clear an Operand
CMP	cmp.S	%dn,EA	Compare
CMPA	cmp.A	%an,EA	Compare Address
CMPI	cmp.S	EA,&I	Compare Immediate
CMPM	cmp.S	(%ax)+, (%ay)+	Compare Memory

MC68020 instruction formats are in **boldface**.
MC68881 instruction formats begin with the letter **f**.

TABLE 15-3

MC68K INSTRUCTION FORMATS (Page 5 of 17)

Operation	Assembler Syntax		Meaning
			Note: The order of operands in CTIX assembly language is the reverse of that in the MC68000 User's Manual.
CMP2	**cmp2.S**	**%rn,EA**	**Compare Register Against Bounds**
DBcc	dbCC	%dn,L	Test Condition, Decrement and Branch
	dbra	%dn,L	Decrement and Branch Always
	dbr	%dn,L	Same as dbra
DIVS	divs.w	EA,%dn	Signed Divide
DIVSL	**divs.l**	**EA,%dq**	**Signed Divide**
	divs.l	**EA,%dr:%dq**	
	divs.l	**EA,%dr:%dq**	
			Note: r represents remainder; q represents quotient.
DIVU	divu.w	EA,%dn	Unsigned Divide
DIVUL	**divu.l**	**EA,%dq**	**Unsigned Divide**
	divu.l	**EA,%dr:%dq**	
	divul.l	**EA,%dr:%dq**	
			Note: r represents remainder; q represents quotient.
EOR	eor.S	%dn,EA	Exclusive-OR Logical

MC68020 instruction formats are in **boldface**.
MC68881 instruction formats begin with the letter **f**.

TABLE 15-3

MC68K INSTRUCTION FORMATS (Page 6 of 17)

Operation	Assembler Syntax		Meaning
EORI	eor.S	&I,EA	Exclusive-OR Immediate
EORI to CCR	eor.b	&I,%cc	Exclusive-OR Immediate to Condition Codes
EORI to SR	eor.w	&I,%sr	Exclusive-OR Immediate to the Status Register (Privileged Instruction)
EXG	exg.l	%rx,%ry	Exchange Registers
EXT	ext.A	%dn	Sign Extend
EXTB	**extb.l**	**%dn**	**Sign Extend**
FABS	fabs.FMT fabs.x fabs.x	EA,%fn %fm,%fn %fn	Absolute Value
FACOS	facos.FMT facos.x facos.x	EA,%fn %fm,%fn %fn	Arc Cosine
FADD	fadd.FMT fadd.x	EA,%fn %fm,%fn	Add
FASIN	fasin.FMT fasin.x fasin.x	EA,%fn %fm,%fn %fn	Arc Sine
FATAN	fatan.FMT fatan.x fatan.x	EA,%fn %fm,%fn %fn	Arc Tangent
FATANH	fatanh.FMT fatanh.x fatanh.x	EA,%fn %fm,%fn %fn	Hyperbolic Arc Tangent

MC68020 instruction formats are in **boldface**.
MC68881 instruction formats begin with the letter **f**.

TABLE 15-3

MC68K INSTRUCTION FORMATS (Page 7 of 17)

Operation	Assembler Syntax		Meaning
FBcc	fbCC.A	L	Branch Conditionally
FCMP	fcmp.FMT	EA,%fn	Compare
	fcmp.x	%fm,%fn	
FCOS	fcos.FMT	EA,%fn	Cosine
	fcos.x	%fm,%fn	
	fcos.x	%fn	
FCOSH	fcosh.FMT	EA,%fn	Hyperbolic Cosine
	fcosh.x	%fm,%fn	
	fcosh.x	%fn	
FDBcc	fdbCC	%dn,L	Test Condition, Decrement and Branch
FDIV	fdiv.FMT	EA,%fn	Divide
	fdiv.x	%fm,%fn	
FETOX	fetox.FMT	EA,%fn	e^x
	fetox.x	%fm,%fn	
	fetox.x	%fn	
FETOXM1	fetoxm1.FMT	EA,%fn	$e^x - 1$
	fetoxm1.x	%fm,%fn	
	fetoxm1.x	%fn	
FGETEXP	fgetexp.FMT	EA,%fn	Get Exponent
	fgetexp.x	%fm,%fn	
	fgetexp.x	%fn	
FGETMAN	fgetman.FMT	EA,%fn	Get Mantissa
	fgetman.x	%fm,%fn	
	fgetman.x	%fn	
FINT	fint.FMT	EA,%fn	Integer Part
	fint.x	%fm,%fn	
	fint.x	%fn	

MC68020 instruction formats are in **boldface**.
MC68881 instruction formats begin with the letter **f**.

TABLE 15-3

MC68K INSTRUCTION FORMATS (Page 8 of 17)

Operation	Assembler Syntax		Meaning
FINTRZ	fintrz.FMT	EA,%fn	Integer Part, Round-to-Zero
	fintrz.x	%fm,%fn	
	fintrz.x	%fn	
FLOG10	flog10.FMT	EA,%fn	Log_{10}
	flog10.x	%fm,%fn	
	flog10.x	%fn	
FLOG2	flog2.FMT	EA,%fn	Log_2
	flog2.x	%fm,%fn	
	flog2.x	%fn	
FLOGN	flogn.FMT	EA,%fn	Log_e
	flogn.x	%fm,%fn	
	flogn.x	%fn	
FLOGNP1	flognp1.FMT	EA,%fn	$Log_e^{(x+1)}$
	flognp1.x	%fm,%fn	
	flognp1.x	%fn	
FMOD	fmod.FMT	EA,%fn	Modulo Remainder
	fmod.x	%fm,%fn	
FMOVE	fmov.FMT	EA,%fn	Move Floating-Point Data Register
	fmov.FMT	%fm,EA	
	fmov.p	%fm,EA{k}	
	fmov.p	%fm,EA{%dn}	
FMOVE	fmov.l	EA,%fpcr	Move System Control Register Note: fp represents one of three control registers. See MC68881 User's Manual for details.
	fmov.l	%fpcr,EA	
FMOVECR	fmovecr.x	&n,%fn	Move Constant ROM

MC68020 instruction formats are in **boldface**.
MC68881 instruction formats begin with the letter **f**.

TABLE 15-3

MC68K INSTRUCTION FORMATS (Page 9 of 17)

Operation	Assembler Syntax		Meaning
			Note: n is offset. See MC68881 User's Manual for details.
FMOVEM	fmovem.x	&I,EA	Move Multiple Data Registers
	fmovem.x	%dn,EA	
	fmovem.x	EA,&I	
	fmovem.x	EA,%dn	
FMOVEM	fmovem.l	&I,EA	Move Multiple Control Registers
	fmovem.l	EA,&I	
			Note: The immediate operand is a mask designating which floating-point registers are to be moved to memory or which floating-point registers are to receive memory data. Not all addressing modes are permitted and the correspondence between mask bits and floating-point register numbers depends on the addressing mode used. See MC68881 User's Manual for details.
FMUL	fmul.FMT	EA,%fn	Multiply
	fmul.x	%fm,%fn	
FNEG	fneg.FMT	EA,%fn	Negate
	fneg.x	%fm,%fn	
	fneg.x	%fn	
FNOP	fnop		No Operation

MC68020 instruction formats are in **boldface**.

MC68881 instruction formats begin with the letter **f**.

TABLE 15-3

MC68K INSTRUCTION FORMATS (Page 10 of 17)

Operation	Assembler Syntax		Meaning
FREM	frem.FMT frem.x	EA,%fn %fm,%fn	IEEE Remainder
FRESTORE	frestore	EA	Restore Internal State (Privileged Instruction)
FSAVE	fsave	EA	Save Internal State (Privileged Instruction)
FSCALE	fscale.FMT fscale.x	EA,%fn %fm,%fn	Scale Exponent
FScc	fbCC.b	EA	Set According to Condition
FSGLDIV	fsgldiv.FMT fsgldiv.x	EA,%fn %fm,%fn	Single Precision Divide
FSGLMUL	fsglmul.FMT fsglmul.x	EA,%fn %fm,%fn	Single Precision Multiply
FSIN	fsin.FMT fsin.x fsin.x	EA,%fn %fm,%fn %fn	Sine
FSINCOS	fsincos.FMT fsincos.x	EA,%fc:%fs %fm,%fc:%fs	Simultaneous Sine and Cosine Note: s represents sine; n represents cosine.
FSINH	fsinh.FMT fsinh.x fsinh.x	EA,%fn %fm,%fn %fn	Hyperbolic Sine
FSQRT	fsqrt.FMT fsqrt.x fsqrt.x	EA,%fn %fm,%fn %fn	Square Root

MC68020 instruction formats are in **boldface**.
MC68881 instruction formats begin with the letter **f**.

TABLE 15-3

MC68K INSTRUCTION FORMATS (Page 11 of 17)

Operation	Assembler Syntax		Meaning
FSUB	fsub.FMT	EA,%fn	Subtract
	fsub.x	%fm,%fn	
FTAN	ftan.FMT	EA,%fn	Tangent
	ftan.x	%fm,%fn	
	ftan.x	%fn	
FTANH	ftanh.FMT	EA,%fn	Hyperbolic Tangent
	ftanh.x	%fm,%fn	
	ftanh.x	%fn	
FTENTOX	ftentox.FMT	EA,%fn	10^x
	ftentox.x	%fm,%fn	
	ftentox.x	%fn	
FTRAPPcc	ftrappCC		Trap Conditionally
	ftrappCC.w	&I	
	ftrappCC.l	&I	
FTST	ftst.FMT	EA	Test Operand
	ftst.x	%fm	
FTWOTOX	ftwotox.FMT	EA,%fn	2^x
	ftwotox.x	%fm,%fn	
	ftwotox.x	%fn	
ILLEGAL	illegal		Take Illegal Instruction Trap
JMP	jmp	EA	Jump
JSR	jsr	EA	Jump to Subroutine
LEA	lea.l	EA,%an	Load Effective Address
LINK	link.w	%an,&I	Link and Allocate
LINK	**link.l**	**%an,&I**	**Link and Allocate**

MC68020 instruction formats are in **boldface**.
MC68881 instruction formats begin with the letter **f**.

TABLE 15-3

MC68K INSTRUCTION FORMATS (Page 12 of 17)

Operation	Assembler Syntax		Meaning
LSL	lsl.S	%dx,%dy &Q,%dy	Logical Shift (Left)
	lsl.w	&I,EA	
LSR	lsr.S	%dx,%dy &Q,%dy	Logical Shift (Right)
	lsr.w	&I,EA	
MOVE	mov.S	EA,EA	Move Data from Source to Destination Note: If the destination is an address register, the instruction generated is MOVEA.
MOVE to CCR	mov.w	EA,%cc	Move to Condition Codes
MOVE to SR	mov.w	EA,%sr	Move to the Status Register (Privileged Instruction)
MOVE from CCR	move.w	%cc,EA	Move From the Condition Code Register
MOVE from SR	mov.w	%sr,EA	Move from the Status Register (Privileged Instruction)
MOVE USP	mov.l	%usp,%an %an,%usp	Move User Stack Pointer (Privileged Instruction if Supervisor State)
MOVEA	mov.A	EA,%an	Move Address
MOVEC	movec.l	%cr,%rn	Move Control Register

MC68020 instruction formats are in **boldface**.
MC68881 instruction formats begin with the letter **f**.

TABLE 15-3

MC68K INSTRUCTION FORMATS (Page 13 of 17)

Operation	Assembler Syntax		Meaning
	movec.l	%rn,%cr	(Privileged Instruction)
MOVEM	movm.A	&I,EA EA,&I	Move Multiple Registers Note: The immediate operand is a mask designating which registers are to be moved to memory or which registers are to receive memory data. Not all addressing modes are permitted, and the correspondence between mask bits and register numbers depends on the addressing mode used. See MC68000 User's Manual for details.
MOVEP	movp.A	%dx,D(%ay)	Move Peripheral Data
MOVEQ	mov.l	&I,%dn	Move Quick (when I fits in one byte)
MOVES	moves.S moves.S	%rn,EA EA,%rn	Move Address Space (Privileged Instruction)
MULS	**muls.l** **muls.l**	**EA,%dl** **EA,%dh:%dl**	**Signed Multiply** **Note: h represents high order; l represents low order.**
MULS	muls.w	EA,%dn	Signed Multiply
MULU	mulu.w	EA,%dn	Unsigned Multiply

MC68020 instruction formats are in **boldface**.
MC68881 instruction formats begin with the letter **f**.

TABLE 15-3

MC68K INSTRUCTION FORMATS (Page 14 of 17)

Operation	Assembler Syntax		Meaning
MULU	**mulu.l** **mulu.l**	**EA,%dl** **EA,%dh:%dl**	**Unsigned Multiply** **Note: h represents high order; l represents low order.**
NBCD	nbcd.b	EA	Negate Decimal with Extend
NEG	neg.S	EA	Negate
NEGX	negx.S	EA	Negate with Extend
NOP	nop		No Operation
NOT	not.S	EA	Logical Complement
OR	or.S	EA,%dn %dn,EA	Inclusive-OR Logical
ORI	or.S	&I,EA	Inclusive-OR Immediate
ORI to CCR	or.b	&I,%cc	Inclusive-OR Immediate to Condition Codes
ORI to SR	or.w	&I,%sr	Inclusive-OR Immediate to the Status Register (Privileged Instruction)
PACK	**pack**	**-(%ax),-(%ay), &number** **%dx,%dy, &number**	**Pack**
PEA	pea.l	EA	Push Effective Address
RESET	reset		Reset External Devices

MC68020 instruction formats are in **boldface**.
MC68881 instruction formats begin with the letter **f**.

TABLE 15-3

MC68K INSTRUCTION FORMATS (Page 15 of 17)

Operation	Assembler Syntax		Meaning
			(Privileged Instruction)
ROL	rol.S	%dx,%dy &Q,%dy	Rotate without Extend (Left)
	rol.w	&I,EA	
ROR	ror.S	%dx,%dy &Q,%dy	Rotate without Extend (Right)
	ror.w	&I,EA	
ROXL	roxl.S	%dx,%dy &Q,%dy	Rotate with Extend (Left)
	roxl.w	&I,EA	
ROXR	roxr.S	%dx,%dy &Q,%dy	Rotate with Extend (Right)
	roxr.w	&I,EA	
RTD	rtd	&number	Return and Deallocate Parameters
RTE	rte		Return from Exception (Privileged Instruction)
RTM	**rtm**	**%rn**	**Return from Module**
RTR	rtr		Return and Restore Condition Codes
RTS	rts		Return from Subroutine
SBCD	sbcd.b	%dy,%dx -(%ay),-(%ax)	Subtract Decimal with Extend

MC68020 instruction formats are in **boldface**.
MC68881 instruction formats begin with the letter **f**.

TABLE 15-3

MC68K INSTRUCTION FORMATS (Page 16 of 17)

Operation	Assembler Syntax		Meaning
Scc	sCC.b	EA	Set According to Condition
STOP	stop	&I	Load Status Register and Stop (Privileged Instruction)
SUB	sub.S	EA,%dn %dn,EA	Subtract Binary
SUBA	sub.A	EA,%an	Subtract Address
SUBI	sub.S	&I,EA	Subtract Immediate
SUBQ	sub.S	&Q,EA	Subtract Quick
SUBX	subx.S	%dy,%dx -(%ay),-(%ax)	Subtract with Extend
SWAP	swap.w	%dn	Swap Register Halves
TAS	tas.b	EA	Test and Set an Operand
TRAP	trap	&I	Trap
TRAPcc	**trapCC** **trapCC.w** **trapCC.l**	 **&I** **&I**	**Trap on Condition**
TRAPV	trapv		Trap on Overflow
TST	tst.S	EA	Test an Operand
UNLK	unlk	%an	Unlink
UNPK	**unpk**	**-(%ax),-(%ay), &number**	**Unpack Binary Coded Decimal**

MC68020 instruction formats are in **boldface**.
MC68881 instruction formats begin with the letter **f**.

TABLE 15-3

MC68K INSTRUCTION FORMATS (Page 17 of 17)

Operation	Assembler Syntax		Meaning
	unpk	%dx,%dy, &number	

MC68020 instruction formats are in **boldface**.

MC68881 instruction formats begin with the letter **f**.

Assembler Differences

CTIX assembler coding differs from the coding conventions used in the MC68K user manuals in two ways.

Comparison Instructions

The CTIX assembler expects to see comparison operands in the reverse order of that presented in the MC68K user manuals. As an example, an MC68K instruction might look like

```
CMP.W     D5,D3          Is D3 less than D5?
BLE       IS_LESS        Branch if less.
```

The equivalent CTIX assembler instruction is

```
cmp.w     %d3,%d5        # Is d3 less than d5?
ble       is_less        # Branch if less.
```

The CTIX assembler follows the convention used by other assemblers supported in the UNIX system (both the 3B 20S computer and the VAX follow this convention). This convention makes for straightforward reading of compare-and-branch instruction sequences. However, it does lead to the peculiarity that if a compare instruction is replaced by a subtract instruction, the effect on the condition codes will be entirely different. This may be confusing to programmers who are used to thinking of a comparison as a subtraction whose result is not stored. Users of the assembler who become accustomed to the convention will find that both the compare and subtract notations make sense in their respective contexts.

Overloading of Opcodes

The CTIX assembler permits operations which do more or less the same thing to be specified by a single assembly instruction. For example, the MC68K user manuals list the instructions **SUB**, **SUBA**, **SUBI**, and **SUBQ**, which all have the effect of subtracting their source operand from their destination operand. The CTIX assembler allows all these operations to be specified by the single assembly instruction **sub**. On the basis of the operands given to the **sub**

instruction, the CTIX assembler selects the appropriate MC68K operation code.

Care must be taken when using this feature, since not all forms of such operations are semantically identical. For example, while **SUB**, **SUBI**, and **SUBQ** all affect the condition codes in a consistent way, **SUBA** does not affect the condition codes at all. Consequently, the CTIX assembler user must be aware that when the destination of a **sub** instruction is an address register (which causes the **sub** to be mapped into the operation code for **SUBA**), the condition codes will not be affected.

Index

D

E

N