

Unix User's Manual Supplementary Documents



Unix User's Manual
Supplementary Documents

USD

**UNIX User's Supplementary Documents
(USD)**

**4.3 Berkeley Software Distribution
Virtual VAX-11 Version**

April, 1986

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

UNIX User's Supplementary Documents (USD)

4.3 Berkeley Software Distribution, Virtual VAX-11 Version

February, 1986

This volume contains documents which supplement the manual pages in *The Unix User's Reference Manual* for the Virtual VAX-11 version of the system as distributed by U.C. Berkeley, and Volumes 2a and 2b as provided by Bell Laboratories.

Getting Started

- Unix for Beginners – Second Edition USD:1
 An introduction to the most basic uses of the system.
- Learn – Computer-Aided Instruction on UNIX (Second Edition) USD:2
 Describes a computer-aided instruction program that walks new users through the basics of files, the editor, and document preparation software.

Basic Utilities

- An Introduction to the UNIX Shell USD:3
 Steve Bourne's introduction to the capabilities of *sh*, a command interpreter especially popular for writing shell scripts.
- An Introduction to the C shell USD:4
 This introduction to *csh*, (a command interpreter popular for interactive work) describes many commonly used UNIX commands, assumes little prior knowledge of UNIX, and has a glossary useful for beginners.
- DC – An Interactive Desk Calculator USD:5
 A super HP calculator, if you do not need floating point.
- BC – An Arbitrary Precision Desk-Calculator Language USD:6
 A front end for DC that provides infix notation, control flow, and built-in functions.

Communicating with the World

- Mail Reference Manual USD:7
 Complete details on one of the programs for sending and reading your mail.
- The Rand MH Message Handling System USD:8
 This system for managing your computer mail uses lots of small programs, instead of one large one.
- How to Read the Network News USD:9
 Describes how news works (generally) and some alternatives for reading it, *readnews* and *vnews*.

A System for Typesetting Mathematics	USD:26
Describes <i>eqn</i> , an easy-to-learn language for high-quality mathematical typesetting.	
Typesetting Mathematics – User's Guide (Second Edition)	USD:27
More details about how to use <i>eqn</i> .	
Tbl – A Program to Format Tables	USD:28
A program for easily typesetting tabular material.	
Refer – A Bibliography System	USD:29
An introduction to one set of tools used to maintain bibliographic databases. The major program, <i>refer</i> , is used to automatically retrieve and format the references based on document citations.	
Some Applications of Inverted Indexes on the UNIX System	USD:30
Mike Lesk's paper describes the <i>refer</i> programs in a somewhat larger context.	
BIB – A Program for Formatting Bibliographies	USD:31
This is an alternative to <i>refer</i> for expanding citations in documents.	
Writing Tools – The STYLE and DICTION Programs	USD:32
These are programs which can help you understand and improve your writing style.	
Amusements	
A Guide to the Dungeons of Doom	USD:33
An introduction to the popular game of <i>rogue</i> , a fantasy game which is one of the biggest known users of VAX cycles.	
Star Trek	USD:34
You are the Captain of the Starship Enterprise. Wipe out the Klingons and save the Federation.	

UNIX For Beginners — Second Edition

Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

(Updated for 4.3BSD by Mark Seiden)

ABSTRACT

This paper is meant to help new users get started on the UNIX† operating system. It includes:

- basics needed for day-to-day use of the system — typing commands, correcting typing mistakes, logging in and out, mail, inter-terminal communication, the file system, printing files, redirecting I/O, pipes, and the shell.
- document preparation — a brief discussion of the major formatting programs and macro packages, hints on preparing documents, and capsule descriptions of some supporting software.
- UNIX programming — using the editor, programming the shell, programming in C, other languages and tools.
- An annotated UNIX bibliography.

INTRODUCTION

From the user's point of view, the UNIX operating system is easy to learn and use, and presents few of the usual impediments to getting the job done. It is hard, however, for the beginner to know where to start, and how to make the best use of the facilities available. The purpose of this introduction is to help new users get used to the main ideas of the UNIX system and start making effective use of it quickly.

You should have a couple of other documents with you for easy reference as you read this one. The most important is *The UNIX Programmer's Manual*; it's often easier to tell you to read about something in the manual than to repeat its contents here. The other useful document is *A Tutorial Introduction to the UNIX Text Editor*, which will tell you how to use the editor to get text — programs, data, documents — into the computer.

A word of warning: the UNIX system has become quite popular, and there are several major variants in widespread use. Of course details also

change with time. So although the basic structure of UNIX and how to use it is common to all versions, there will certainly be a few things which are different on your system from what is described here. We have tried to minimize the problem, but be aware of it. In cases of doubt, this paper describes Version 7 UNIX.

This paper has five sections:

1. Getting Started: How to log in, how to type, what to do about mistakes in typing, how to log out. Some of this is dependent on which system you log into (phone numbers, for example) and what terminal you use, so this section must necessarily be supplemented by local information.
2. Day-to-day Use: Things you need every day to use the system effectively: generally useful commands; the file system.
3. Document Preparation: Preparing manuscripts is one of the most common uses for UNIX systems. This section contains advice, but not extensive instructions on any of the formatting

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

command

stty -tabs

and the system will convert each tab into the right number of blanks for you. If your terminal does have computer-settable tabs, the command **tabs** will set the stops correctly for you.

Mistakes in Typing

If you make a typing mistake, and see it before RETURN has been typed, there are two ways to recover. The sharp-character # erases the last character typed; in fact successive uses of # erase characters back to the beginning of the line (but not beyond). So if you type badly, you can correct as you go:

dd#atte##e

is the same as **date**.‡

The at-sign @ erases all of the characters typed so far on the current input line, so if the line is irretrievably fouled up, type an @ and start the line over.

What if you must enter a sharp or at-sign as part of the text? If you precede either # or @ by a backslash \, it loses its erase meaning. So to enter a sharp or at-sign in something, type \# or \@. The system will always echo a newline at you after your at-sign, even if preceded by a backslash. Don't worry — the at-sign has been recorded.

To erase a backslash, you have to type two sharps or two at-signs, as in \##. The backslash is used extensively in UNIX to indicate that the following character is in some way special.

Read-ahead

UNIX has full read-ahead, which means that you can type as fast as you want, whenever you want, even when some command is typing at you. If you type during output, your input characters will appear intermixed with the output characters, but they will be stored away and interpreted in the correct order. So you can type several commands one after another without waiting for the first to finish or even begin.

Stopping a Program

You can stop most programs by typing the character "DEL" (perhaps called "delete" or "rubout" on your terminal). The "interrupt" or "break" key found on most terminals can also be used.† In a few programs, like the text editor, DEL stops whatever

‡ Many installations set the erase character for display terminals to the delete or backspace key. "stty all" tells you what it actually is.

† In Berkeley Unix, "control-c" is the usual way to stop programs. "stty all" tells you the value of your "intr" key.

the program is doing but leaves you in that program. Hanging up the phone will stop most programs.‡

Logging Out

The easiest way to log out is to hang up the phone. You can also type

login

and let someone else use the terminal you were on.* It is usually not sufficient just to turn off the terminal. Most UNIX systems do not use a time-out mechanism, so you'll be there forever unless you hang up.

Mail

When you log in, you may sometimes get the message

You have mail.

UNIX provides a postal system so you can communicate with other users of the system. To read your mail, type the command

mail

Your mail will be printed, one message at a time, most recent message first.‡ After each message, **mail** waits for you to say what to do with it. The two basic responses are **d**, which deletes the message, and RETURN, which does not (so it will still be there the next time you read your mailbox). Other responses are described in the manual. (Earlier versions of **mail** do not process one message at a time, but are otherwise similar.)

How do you send mail to someone else? Suppose it is to go to "joe" (assuming "joe" is someone's login name). The easiest way is this:

mail joe

now type in the text of the letter

on as many lines as you like ...

After the last line of the letter

type the character "control-d",

that is, hold down "control" and type

a letter "d".

And that's it. The "control-d" sequence, often called "EOF" for end-of-file, is used throughout the system to mark the end of input from a terminal, so you might as well get used to it.

For practice, send mail to yourself. (This isn't as strange as it might sound — mail to oneself is a handy reminder mechanism.)

‡ If you use the **c** shell, programs running in the background continue running even if you hang up.

* "control-d" and "logout" are other alternatives.

‡ The Berkeley mail program lists the headers of some number of unread pieces of mail in the order of their receipt.

have typed into a file with the editor command w:

w

ed will respond with the number of characters it wrote into the file junk.

Until the w command, nothing is stored permanently, so if you hang up and go home the information is lost.† But after w the information is there permanently; you can re-access it any time by typing

ed junk

Type a q command to quit the editor. (If you try to quit without writing, ed will print a ? to remind you. A second q gets you out regardless.)

Now create a second file called temp in the same manner. You should now have two files, junk and temp.

What files are out there?

The ls (for "list") command lists the names (not contents) of any of the files that UNIX knows about. If you type

ls

the response will be

```
junk
temp
```

which are indeed the two files just created. The names are sorted into alphabetical order automatically, but other variations are possible. For example, the command

ls -t

causes the files to be listed in the order in which they were last changed, most recent first. The -l option gives a "long" listing:

ls -l

will produce something like

```
-rw-rw-rw- 1 bwk 41 Jul 22 2:56 junk
-rw-rw-rw- 1 bwk 78 Jul 22 2:57 temp
```

The date and time are of the last change to the file. The 41 and 78 are the number of characters (which should agree with the numbers you got from ed). bwk is the owner of the file, that is, the person who created it. The -rw-rw-rw- tells who has permission to read and write the file, in this case everyone.

Options can be combined: ls -lt gives the same thing as ls -l, but sorted into time order. You can also name the files you're interested in, and ls will list the information about them only. More details

† This is not strictly true — if you hang up while editing, the data you were working on is saved in a file called ed.hup, which you can continue with at your next session.

can be found in ls(1).

The use of optional arguments that begin with a minus sign, like -t and -lt, is a common convention for UNIX programs. In general, if a program accepts such optional arguments, they precede any filename arguments. It is also vital that you separate the various arguments with spaces: ls-l is not the same as ls -l.

Printing Files

Now that you've got a file of text, how do you print it so people can look at it? There are a host of programs that do that, probably more than are needed.

One simple thing is to use the editor, since printing is often done just before making changes anyway. You can say

```
ed junk
1,$p
```

ed will reply with the count of the characters in junk and then print all the lines in the file. After you learn how to use the editor, you can be selective about the parts you print.

There are times when it's not feasible to use the editor for printing. For example, there is a limit on how big a file ed can handle (several thousand lines). Secondly, it will only print one file at a time, and sometimes you want to print several, one after another. So here are a couple of alternatives.

First is cat, the simplest of all the printing programs. cat simply prints on the terminal the contents of all the files named in a list. Thus

```
cat junk
```

prints one file, and

```
cat junk temp
```

prints two. The files are simply concatenated (hence the name "cat") onto the terminal.

pr produces formatted printouts of files. As with cat, pr prints all the files named in a list. The difference is that it produces headings with date, time, page number and file name at the top of each page, and extra lines to skip over the fold in the paper. Thus,

```
pr junk temp
```

will print junk neatly, then skip to the top of a new page and print temp neatly.

pr can also produce multi-column output:

```
pr -3 junk
```

prints junk in 3-column format. You can use any reasonable number in place of "3" and pr will do its best. pr has other capabilities as well; see pr(1).

```

chap1.1
chap1.2
chap1.3
...

```

The * is not limited to the last position in a filename — it can be anywhere and can occur several times. Thus

```
rm *junk* *temp*
```

removes all files that contain **junk** or **temp** as any part of their name. As a special case, * by itself matches every filename, so

```
pr *
```

prints all your files (alphabetical order), and

```
rm *
```

removes *all files*. (You had better be *very* sure that's what you wanted to say!)

The * is not the only pattern-matching feature available. Suppose you want to print only chapters 1 through 4 and 9. Then you can say

```
pr chap[12349]*
```

The [...] means to match any of the characters inside the brackets. A range of consecutive letters or digits can be abbreviated, so you can also do this with

```
pr chap[1-49]*
```

Letters can also be used within brackets: [a-z] matches any character in the range a through z.

The ? pattern matches any single character, so

```
ls ?
```

lists all files which have single-character names, and

```
ls -l chap?.1
```

lists information about the first file of each chapter (**chap1.1**, **chap2.1**, etc.).

Of these niceties, * is certainly the most useful, and you should get used to it. The others are frills, but worth knowing.

If you should ever have to turn off the special meaning of *, ?, etc., enclose the entire argument in single quotes, as in

```
ls '?'
```

We'll see some more examples of this shortly.

What's in a Filename, Continued

When you first made that file called **junk**, how did the system know that there wasn't another **junk** somewhere else, especially since the person in the next office is also reading this tutorial? The answer is that generally each user has a private *directory*, which contains only the files that belong to him.

When you log in, you are "in" your directory. Unless you take special action, when you create a new file, it is made in the directory that you are currently in; this is most often your own directory, and thus the file is unrelated to any other file of the same name that might exist in someone else's directory.

The set of all files is organized into a (usually big) tree, with your files located several branches into the tree. It is possible for you to "walk" around this tree, and to find any file in the system, by starting at the root of the tree and walking along the proper set of branches. Conversely, you can start where you are and walk toward the root.

Let's try the latter first. The basic tool is the command **pwd** ("print working directory"), which prints the name of the directory you are currently in.

Although the details will vary according to the system you are on, if you give the command **pwd**, it will print something like

```
/usr/your-name
```

This says that you are currently in the directory **your-name**, which is in turn in the directory **/usr**, which is in turn in the root directory called by convention just **/**. (Even if it's not called **/usr** on your system, you will get something analogous. Make the corresponding mental adjustment and read on.)

If you now type

```
ls /usr/your-name
```

you should get exactly the same list of file names as you get from a plain **ls**: with no arguments, **ls** lists the contents of the current directory; given the name of a directory, it lists the contents of that directory.

Next, try

```
ls /usr
```

This should print a long series of names, among which is your own login name **your-name**. On many systems, **usr** is a directory that contains the directories of all the normal users of the system, like you.

The next step is to try

```
ls /
```

You should get a response something like this (although again the details may be different):

```

bin
dev
etc
lib
tmp
usr

```

This is a collection of the basic directories of files that the system knows about; we are at the root of the tree.

put. As one example,

```
ls
```

makes a list of files on your terminal. But if you say

```
ls >filelist
```

a list of your files will be placed in the file `filelist` (which will be created if it doesn't already exist, or overwritten if it does). The symbol `>` means "put the output on the following file, rather than on the terminal." Nothing is produced on the terminal. As another example, you could combine several files into one by capturing the output of `cat` in a file:

```
cat f1 f2 f3 >temp
```

The symbol `>>` operates very much like `>` does, except that it means "add to the end of." That is,

```
cat f1 f2 f3 >>temp
```

means to concatenate `f1`, `f2` and `f3` to the end of whatever is already in `temp`, instead of overwriting the existing contents. As with `>`, if `temp` doesn't exist, it will be created for you.

In a similar way, the symbol `<` means to take the input for a program from the following file, instead of from the terminal. Thus, you could make up a script of commonly used editing commands and put them into a file called `script`. Then you can run the script on a file by saying

```
ed file <script
```

As another example, you can use `ed` to prepare a letter in file `let`, then send it to several people with

```
mail adam eve mary joe <let
```

Pipes

One of the novel contributions of the UNIX system is the idea of a *pipe*. A pipe is simply a way to connect the output of one program to the input of another program, so the two run as a sequence of processes — a pipeline.

For example,

```
pr f g h
```

will print the files `f`, `g`, and `h`, beginning each on a new page. Suppose you want them run together instead. You could say

```
cat f g h >temp
pr <temp
rm temp
```

but this is more work than necessary. Clearly what we want is to take the output of `cat` and connect it to the input of `pr`. So let us use a pipe:

```
cat f g h | pr
```

The vertical bar `|` means to take the output from `cat`, which would normally have gone to the terminal, and put it into `pr` to be neatly formatted.

There are many other examples of pipes. For example,

```
ls | pr -3
```

prints a list of your files in three columns. The program `we` counts the number of lines, words and characters in its input, and as we saw earlier, `who` prints a list of currently-logged on people, one per line. Thus

```
who | wc
```

tells how many people are logged on. And of course

```
ls | wc
```

counts your files.

Any program that reads from the terminal can read from a pipe instead; any program that writes on the terminal can drive a pipe. You can have as many elements in a pipeline as you wish.

Many UNIX programs are written so that they will take their input from one or more files if file arguments are given; if no arguments are given they will read from the terminal, and thus can be used in pipelines. `pr` is one example:

```
pr -3 a b c
```

prints files `a`, `b` and `c` in order in three columns. But in

```
cat a b c | pr -3
```

`pr` prints the information coming down the pipeline, still in three columns.

The Shell

We have already mentioned once or twice the mysterious "shell," which is in fact `sh(1)`.† The shell is the program that interprets what you type as commands and arguments. It also looks after translating `*`, etc., into lists of filenames, and `<`, `>`, and `|` into changes of input and output streams.

The shell has other capabilities too. For example, you can run two programs with one command line by separating the commands with a semicolon; the shell recognizes the semicolon and breaks the line into two commands. Thus

```
date; who
```

does both commands before returning with a prompt character.

† On Berkeley Unix systems, the usual shell for interactive use is the `c` shell, `esh(1)`.

.TL
title of document
.AU
author name
.SH
section heading
.PP
paragraph ...
.PP
another paragraph ...
.SH
another section heading
.PP
etc.

The lines that begin with a period are the formatting requests. For example, **.PP** calls for starting a new paragraph. The precise meaning of **.PP** depends on what output device is being used (typesetter or terminal, for instance), and on what publication the document will appear in. For example, **-ms** normally assumes that a paragraph is preceded by a space (one line in **nroff**, 1/2 line in **troff**), and the first word is indented. These rules can be changed if you like, but they are changed by changing the interpretation of **.PP**, not by re-typing the document.

To actually produce a document in standard format using **-ms**, use the command

troff -ms files ...

for the typesetter, and

nroff -ms files ...

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

There are several similar packages; check with a local expert to determine which ones are in common use on your machine.

Supporting Tools

In addition to the basic formatters, there is a host of supporting programs that help with document preparation. The list in the next few paragraphs is far from complete, so browse through the manual and check with people around you for other possibilities.

eqn and **neqn** let you integrate mathematics into the text of a document, in an easy-to-learn language that closely resembles the way you would speak it aloud. For example, the **eqn** input

sum from i=0 to n x sub i ~ = pi over 2

produces the output

$$\sum_{i=0}^n x_i = \frac{\pi}{2}$$

The program **tbl** provides an analogous service for preparing tabular material; it does all the computations necessary to align complicated columns with elements of varying widths.

refer prepares bibliographic citations from a data base, in whatever style is defined by the formatting package. It looks after all the details of numbering references in sequence, filling in page and volume numbers, getting the author's initials and the journal name right, and so on.

spell and **typo** detect possible spelling mistakes in a document.† **spell** works by comparing the words in your document to a dictionary, printing those that are not in the dictionary. It knows enough about English spelling to detect plurals and the like, so it does a very good job. **typo** looks for words which are "unusual", and prints those. Spelling mistakes tend to be more unusual, and thus show up early when the most unusual words are printed first.

grep looks through a set of files for lines that contain a particular text pattern (rather like the editor's context search does, but on a bunch of files). For example,

grep 'ing\$' chap*

will find all lines that end with the letters **ing** in the files **chap***. (It is almost always a good practice to put single quotes around the pattern you're searching for, in case it contains characters like ***** or **\$** that have a special meaning to the shell.) **grep** is often useful for finding out in which of a set of files the misspelled words detected by **spell** are actually located.

diff prints a list of the differences between two files, so you can compare two versions of something automatically (which certainly beats proofreading by hand).

wc counts the words, lines and characters in a set of files. **tr** translates characters into other characters; for example it will convert upper to lower case and vice versa. This translates upper into lower:

tr A-Z a-z <input >output

sort sorts files in a variety of ways; **cref** makes cross-references; **ptx** makes a permuted index (keyword-in-context listing). **sed** provides many of the editing facilities of **ed**, but can apply them to arbitrarily long inputs. **awk** provides the ability to do both pattern matching and numeric computations, and to conveniently process fields within lines. These programs are for more advanced users, and they are not limited to document preparation. Put them on your list of things to learn about.

† "typo" is not provided with Berkeley Unix.

routines, and interrupt handling. Since there are many building-block programs, you can sometimes avoid writing a new program merely by piecing together some of the building blocks with shell command files.

We will not go into any details here; examples and rules can be found in *An Introduction to the UNIX Shell*, by S. R. Bourne.

Programming in C

If you are undertaking anything substantial, C is the only reasonable choice of programming language: everything in the UNIX system is tuned to it. The system itself is written in C, as are most of the programs that run on it. It is also a easy language to use once you get started. C is introduced and fully described in *The C Programming Language* by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978). Several sections of the manual describe the system interfaces, that is, how you do I/O and similar functions. Read *UNIX Programming* for more complicated things.

Most input and output in C is best handled with the standard I/O library, which provides a set of I/O functions that exist in compatible form on most machines that have C compilers. In general, it's wisest to confine the system interactions in a program to the facilities provided by this library.

C programs that don't depend too much on special features of UNIX (such as pipes) can be moved to other computers that have C compilers. The list of such machines grows daily; in addition to the original PDP-11, it currently includes at least Honeywell 6000, IBM 370 and PC families, Interdata 8/32, Data General Nova and Eclipse, HP 2100, Harris /7, Motorola 68000 family (including machines like Sun Microsystems and Apple Macintosh), VAX 11 family, SEL 86, and Zilog Z80. Calls to the standard I/O library will work on all of these machines.

There are a number of supporting programs that go with C. `lint` checks C programs for potential portability problems, and detects errors such as mismatched argument types and uninitialized variables.

For larger programs (anything whose source is on more than one file) `make` allows you to specify the dependencies among the source files and the processing steps needed to make a new version; it then checks the times that the pieces were last changed and does the minimal amount of recompiling to create a consistent updated version.

The debugger `adb` is useful for digging through the dead bodies of C programs, but is rather hard to learn to use effectively. The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.†

† The "dbx" debugger, supplied starting with 4.2BSD, has extensive facilities for high-level debugging of C programs and is

The C compiler provides a limited instrumentation service, so you can find out where programs spend their time and what parts are worth optimizing. Compile the routines with the `-p` option; after the test run, use `prof` to print an execution profile. The command `time` will give you the gross run-time statistics of a program, but they are not super accurate or reproducible.

Other Languages

If you *have* to use Fortran, there are two possibilities. You might consider Ratfor, which gives you the decent control structures and free-form input that characterize C, yet lets you write code that is still portable to other environments. Bear in mind that UNIX Fortran tends to produce large and relatively slow-running programs. Furthermore, supporting software like `adb`, `prof`, etc., are all virtually useless with Fortran programs. There may also be a Fortran 77 compiler on your system. If so, this is a viable alternative to Ratfor, and has the non-trivial advantage that it is compatible with C and related programs. (The Ratfor processor and C tools can be used with Fortran 77 too.)

If your application requires you to translate a language into a set of actions or another language, you are in effect building a compiler, though probably a small one. In that case, you should be using the `yacc` compiler-compiler, which helps you develop a compiler quickly. The `lex` lexical analyzer generator does the same job for the simpler languages that can be expressed as regular expressions. It can be used by itself, or as a front end to recognize inputs for a `yacc`-based program. Both `yacc` and `lex` require some sophistication to use, but the initial effort of learning them can be repaid many times over in programs that are easy to change later on.

Most UNIX systems also make available other languages, such as Algol 68, APL, Basic, Lisp, Pascal, and Snobol. Whether these are useful depends largely on the local environment: if someone cares about the language and has worked on it, it may be in good shape. If not, the odds are strong that it will be more trouble than it's worth.

V. UNIX READING LIST

General:

K. L. Thompson and D. M. Ritchie, *The UNIX Programmer's Manual*, Bell Laboratories, 1978 (PS2:3)‡ Lists commands, system routines and interfaces, file formats, and some of the maintenance procedures. You can't live without this, although you will probably only need to read section 1.

‡ much easier to use than "adb".

LEARN — Computer-Aided Instruction on UNIX (Second Edition)

Brian W. Kernighan

Michael E. Lesk

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes the second version of the *learn* program for interpreting CAI scripts on the UNIX† operating system, and a set of scripts that provide a computerized introduction to the system.

Six current scripts cover basic commands and file handling, the editor, additional file handling commands, the *eqn* program for mathematical typing, the “-ms” package of formatting macros, and an introduction to the C programming language. These scripts now include a total of about 530 lessons.

Many users from a wide variety of backgrounds have used *learn* to acquire basic UNIX skills. Most usage involves the first two scripts, an introduction to UNIX files and commands, and the UNIX editor.

The second version of *learn* is about four times faster than the previous one in CPU utilization, and much faster in perceived time because of better overlap of computing and printing. It also requires less file space than the first version. Many of the lessons have been revised; new material has been added to reflect changes and enhancements in UNIX itself. Script-writing is also easier because of revisions to the script language.

1. Introduction.

Learn is a driver for CAI scripts. It is intended to permit the easy composition of lessons and lesson fragments to teach people computer skills. Since it is teaching the same system on which it is implemented, it makes direct use of UNIX facilities to create a controlled UNIX environment. The system includes two main parts: (1) a driver that interprets the lesson scripts; and (2) the lesson scripts themselves. At present there are seven scripts:

- basic file handling commands
- the UNIX text editors *ed* and *vi*
- advanced file handling
- the *eqn* language for typing mathematics
- the “ms” macro package for document formatting
- the C programming language

The purported advantages of CAI scripts for training in computer skills include the following:

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

Figure 1: Sample dialog from basic files script

(Student responses in italics; '\$' is the prompt)

A file can be printed on your terminal by using the "cat" command. Just say "cat file" where "file" is the file name. For example, there is a file named "food" in this directory. List it by saying "cat food"; then type "ready".

```
$ cat food
  this is the file
  named food.
$ ready
```

Good. Lesson 3.3a (1)

Of course, you can print any file with "cat". In particular, it is common to first use "ls" to find the name of a file and then "cat" to print it. Note the difference between "ls", which tells you the name of the file, and "cat", which tells you the contents. One file in the current directory is named for a President. Print the file, then type "ready".

```
$ cat President
cat: can't open President
$ ready
```

Sorry, that's not right. Do you want to try again? *yes*
Try the problem again.

```
$ ls
.ocopy
X1
roosevelt
$ cat roosevelt
  this file is named roosevelt
  and contains three lines of
  text.
$ ready
```

Good. Lesson 3.3b (0)

The "cat" command can also print several files at once. In fact, it is named "cat" as an abbreviation for "concatenate"....

the answer is wrong, the student is offered a chance to repeat the lesson. The "speed" rating of the student (explained in section 5) is given after the lesson number when the lesson is completed successfully; it is printed only for the aid of script authors checking out possible errors in the lessons.

It is assumed that there is no foolproof way to determine if the student truly "understands" what he or she is doing; accordingly, the current *learn* scripts only measure performance, not comprehension. If the student can perform a given task, that is deemed to be "learning."⁴

detected, the easier it will be on the student. Anyone proceeding through the scripts should be getting mostly correct answers; otherwise, the system will be unsatisfactory both because the wrong habits are being learned and because the scripts make little effort to deal with wrong answers. Unprepared students should not be encouraged to continue with scripts.

There are some preliminary items which the student must know before any scripts can be tried. In particular, the student must know how to connect to a UNIX system, set the terminal properly, log in, and execute simple commands (e.g., *learn* itself). In addition, the character erase and line kill conventions (# and @) should be known. It is hard to see how this much could be taught by computer-aided instruction, since a student who does not know these basic skills will not be able to run the learning program. A brief description on paper is provided (see Appendix A), although assistance will be needed for the first few minutes. This assistance, however, need not be highly skilled.

The first script in the current set deals with files. It assumes the basic knowledge above and teaches the student about the *ls*, *cat*, *mv*, *rm*, *cp* and *diff* commands. It also deals with the abbreviation characters *, ?, and [] in file names. It does not cover pipes or I/O redirection, nor does it present the many options on the *ls* command.

This script contains 31 lessons in the fast track; two are intended as prerequisite checks, seven are review exercises. There are a total of 75 lessons in all three tracks, and the instructional passages typed at the student to begin each lesson total 4,476 words. The average lesson thus begins with a 60-word message. In general, the fast track lessons have somewhat longer introductions, and the slow tracks somewhat shorter ones. The longest message is 144 words and the shortest 14.

The second script trains students in the use of the UNIX context editor *ed*, a sophisticated editor using regular expressions for searching.⁵ All editor features except encryption, mark names and ';' in addressing are covered. The fast track contains 2 prerequisite checks, 93 lessons, and a review lesson. It is supplemented by 146 additional lessons in other tracks.

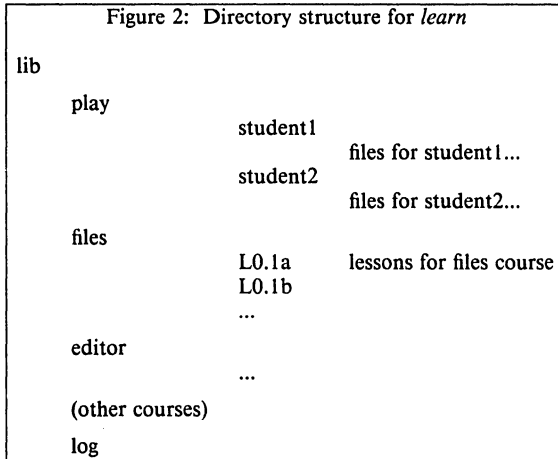
A comparison of sizes may be of interest. The *ed* description in the reference manual is 2,572 words long. The *ed* tutorial⁶ is 6,138 words long. The fast track through the *ed* script is 7,407 words of explanatory messages, and the total *ed* script, 242 lessons, has 15,615 words. The average *ed* lesson is thus also about 60 words; the largest is 171 words and the smallest 10. The original *ed* script represents about three man-weeks of effort.

The advanced file handling script deals with *ls* options, I/O diversion, pipes, and supporting programs like *pr*, *wc*, *tail*, *spell* and *grep*. (The basic file handling script is a prerequisite.) It is not as refined as the first two scripts; this is reflected at least partly in the fact that it provides much less of a full three-track sequence than they do. On the other hand, since it is perceived as "advanced," it is hoped that the student will have somewhat more sophistication and be better able to cope with it at a reasonably high level of performance.

A fourth script covers the *eqn* language for typing mathematics. This script must be run on a terminal capable of printing mathematics, for instance the DASI 300 and similar Diablo-based terminals, or the nearly extinct Model 37 teletype. Again, this script is relatively short of tracks: of 76 lessons, only 17 are in the second track and 2 in the third track. Most of these provide additional practice for students who are having trouble in the first track.

The *-ms* script for formatting macros is a short one-track only script. The macro package it describes is no longer the standard, so this script will undoubtedly be superseded in the future. Furthermore, the linear style of a single learn script is somewhat inappropriate for the macros, since the macro package is composed of many independent features, and few users need all of them. It would be better to have a selection of short lesson sequences dealing with the features independently.

The script on C is in a state of transition. It was originally designed to follow a tutorial on C, but that document has since become obsolete. The current script has been partially converted to follow the order of presentation in *The C Programming Language*,⁷ but this job is not complete. The C script was never intended to teach C; rather it is supposed to be a series of exercises for which the computer provides checking and (upon success) a suggested solution.



(5) a list of possible successor lessons.

Learn tries to minimize the work of bookkeeping and installation, so that most of the effort involved in script production is in planning lessons, writing tutorial paragraphs, and coding tests of student performance.

The basic sequence of events is as follows. First, *learn* creates the working directory. Then, for each lesson, *learn* reads the script for the lesson and processes it a line at a time. The lines in the script are: (1) commands to the script interpreter to print something, to create a file, to test something, etc.; (2) text to be printed or put in a file; (3) other lines, which are sent to the shell to be executed. One line in each lesson turns control over to the user; the user can run any UNIX commands. The user mode terminates when the user types *yes*, *no*, *ready*, or *answer*. At this point, the user's work is tested; if the lesson is passed, a new lesson is selected, and if not the old one is repeated.

Let us illustrate this with the script for the second lesson of Figure 1; this is shown in Figure 3. Lines which begin with # are commands to the *learn* script interpreter. For example,

```
#print
```

causes printing of any text that follows, up to the next line that begins with a sharp.

```
#print file
```

prints the contents of *file*; it is the same as *cat file* but has less overhead. Both forms of *#print* have the added property that if a lesson is failed, the *#print* will not be executed the second time through; this avoids annoying the student by repeating the preamble to a lesson.

```
#create filename
```

creates a file of the specified name, and copies any subsequent text up to a # to the file. This is used for creating and initializing working files and reference data for the lessons.

```
#user
```

gives control to the student; each line he or she types is passed to the shell for execution. The *#user* mode is terminated when the student types one of *yes*, *no*, *ready* or *answer*. At that time, the driver resumes interpretation of the script.

```
#copyin
```

```
#uncopyin
```

Figure 4: Another Sample Lesson

```
#print
What command will move the current line
to the end of the file? Type
"answer COMMAND", where COMMAND is the command.
#copyin
#user
#uncopyin
#match m$
#match .m$
"m$" is easier.
#log
#next
63.1d 10
```

This is similar to `#match`, except that it corresponds to specific failure answers; this can be used to produce hints for particular wrong answers that have been anticipated by the script writer.

```
#succeed
#fail
```

print a message upon success or failure (as determined by some previous mechanism).

When the student types one of the "commands" *yes*, *no*, *ready*, or *answer*, the driver terminates the `#user` command, and evaluation of the student's work can begin. This can be done either by the built-in commands above, such as `#match` and `#cmp`, or by status returned by normal UNIX commands, typically `grep` and `test`. The last command should return status true (0) if the task was done successfully and false (non-zero) otherwise; this status return tells the driver whether or not the student has successfully passed the lesson.

Performance can be logged:

```
#log file
```

writes the date, lesson, user name and speed rating, and a success/failure indication on *file*. The command

```
#log
```

by itself writes the logging information in the logging directory within the *learn* hierarchy, and is the normal form.

```
#next
```

is followed by a few lines, each with a successor lesson name and an optional speed rating on it. A typical set might read

```
25.1a 10
25.2a 5
25.3a 2
```

indicating that unit 25.1a is a suitable follow-on lesson for students with a speed rating of 10 units, 25.2a for student with speed near 5, and 25.3a for speed near 2. Speed ratings are maintained for each session with a student; the rating is increased by one each time the student gets a lesson right and decreased by four each time the student gets a lesson wrong. Thus the driver tries to maintain a level such that the users get 80% right answers. The maximum rating is limited to 10 and the minimum to 0. The initial rating is zero unless the student specifies a different rating when starting a session.



computer. They should exercise the scripts on the same computer and the same kind of terminal that they will later use for their real work, and their first few jobs for the computer should be relatively easy ones. Also, both training and initial work should take place on days when the UNIX hardware and software are working reliably. Rarely is all of this possible, but the closer one comes the better the result. For example, if it is known that the hardware is shaky one day, it is better to attempt to reschedule training for another one. Students are very frustrated by machine downtime; when nothing is happening, it takes some sophistication and experience to distinguish an infinite loop, a slow but functioning program, a program waiting for the user, and a broken machine.*

One disadvantage of training with *learn* is that students come to depend completely on the CAI system, and do not try to read manuals or use other learning aids. This is unfortunate, not only because of the increased demands for completeness and accuracy of the scripts, but because the scripts do not cover all of the UNIX system. New users should have manuals (appropriate for their level) and read them; the scripts ought to be altered to recommend suitable documents and urge students to read them.

There are several other difficulties which are clearly evident. From the student's viewpoint, the most serious is that lessons still crop up which simply can't be passed. Sometimes this is due to poor explanations, but just as often it is some error in the lesson itself — a botched setup, a missing file, an invalid test for correctness, or some system facility that doesn't work on the local system in the same way it did on the development system. It takes knowledge and a certain healthy arrogance on the part of the user to recognize that the fault is not his or hers, but the script writer's. Permitting the student to get on with the next lesson regardless does alleviate this somewhat, and the logging facilities make it easy to watch for lessons that no one can pass, but it is still a problem.

The biggest problem with the previous *learn* was speed (or lack thereof) — it was often excruciatingly slow and made a significant drain on the system. The current version so far does not seem to have that difficulty, although some scripts, notably *eqn*, are intrinsically slow. *eqn*, for example, must do a lot of work even to print its introductions, let alone check the student responses, but delay is perceptible in all scripts from time to time.

Another potential problem is that it is possible to break *learn* inadvertently, by pushing interrupt at the wrong time, or by removing critical files, or any number of similar slips. The defenses against such problems have steadily been improved, to the point where most students should not notice difficulties. Of course, it will always be possible to break *learn* maliciously, but this is not likely to be a problem.

One area is more fundamental — some UNIX commands are sufficiently global in their effect that *learn* currently does not allow them to be executed at all. The most obvious is *cd*, which changes to another directory. The prospect of a student who is learning about directories inadvertently moving to some random directory and removing files has deterred us from even writing lessons on *cd*, but ultimately lessons on such topics probably should be added.

7. Acknowledgments

We are grateful to all those who have tried *learn*, for we have benefited greatly from their suggestions and criticisms. In particular, M. E. Bittrich, J. L. Blue, S. I. Feldman, P. A. Fox, and M. J. McAlpin have provided substantial feedback. Conversations with E. Z. Rothkopf also provided many of the ideas in the system. We are also indebted to Don Jackowski for serving as a guinea pig for the second version, and to Tom Plum for his efforts to improve the C script.

References

1. D.L. Bitzer and D. Skaperdas, "The Economics of a Large Scale Computer Based Educational System: Plato IV," in *Computer Assisted Instruction, Testing and Guidance*, ed. Wayne Holtzman, pp. 17-29, Harper and Row, New York, 1970.

* We have even known an expert programmer to decide the computer was broken when he had simply left his terminal in local mode. Novices have great difficulties with such problems.

An Introduction to the UNIX Shell

S. R. Bourne

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

(Updated for 4.3BSD by Mark Seiden)

ABSTRACT

The *shell*‡ is a command programming language that provides an interface to the UNIX† operating system. Its features include control-flow primitives, parameter passing, variables and string substitution. Constructs such as *while*, *if then else*, *case* and *for* are available. Two-way communication is possible between the *shell* and commands. String-valued parameters, typically file names or flags, may be passed to a command. A return code is set by commands that may be used to determine control-flow, and the standard output from a command may be used as shell input.

The *shell* can modify the environment in which commands run. Input and output can be redirected to files, and processes that communicate through 'pipes' can be invoked. Commands are found by searching directories in the file system in a sequence that can be defined by the user. Commands can be read either from the terminal or from a file, which allows command procedures to be stored for later use.

1.0 Introduction

The shell is both a command language and a programming language that provides an interface to the UNIX operating system. This memorandum describes, with examples, the UNIX shell. The first section covers most of the everyday requirements of terminal users. Some familiarity with UNIX is an advantage when reading this section; see, for example, "UNIX for beginners".¹ Section 2 describes those features of the shell primarily intended for use within shell procedures. These include the control-flow primitives and string-valued variables provided by the shell. A knowledge of a programming language would be a help when reading this section. The last section describes the more advanced features of the shell. References of the form "see *pipe* (2)" are to a section of the UNIX manual.²

1.1 Simple commands

Simple commands consist of one or more words separated by blanks. The first word is the name of the command to be executed; any remaining words are passed as arguments to the command. For example,

```
who
```

is a command that prints the names of users logged in. The command

‡ This paper describes sh(1). If it's the c shell (csh) you're interested in, a good place to begin is William Joy's paper "An Introduction to the C shell" (USD:4).

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



For example,

```
who | sort
```

will print an alphabetically sorted list of logged in users.

A pipeline may consist of more than two commands, for example,

```
ls | grep old | wc -l
```

prints the number of file names in the current directory containing the string *old*.

1.5 File name generation

Many commands accept arguments which are file names. For example,

```
ls -l main.c
```

prints information relating to the file *main.c*.

The shell provides a mechanism for generating a list of file names that match a pattern. For example,

```
ls -l *.c
```

generates, as arguments to *ls*, all file names in the current directory that end in *.c*. The character *** is a pattern that will match any string including the null string. In general *patterns* are specified as follows.

- * Matches any string of characters including the null string.
- ? Matches any single character.
- [...] Matches any one of the characters enclosed. A pair of characters separated by a minus will match any character lexically between the pair.

For example,

```
[a-z]*
```

matches all names in the current directory beginning with one of the letters *a* through *z*.

```
/usr/fred/test/?
```

matches all names in the directory */usr/fred/test* that consist of a single character. If no file name is found that matches the pattern then the pattern is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

```
echo /usr/fred/*/core
```

finds and prints the names of all *core* files in sub-directories of */usr/fred*. (*echo* is a standard UNIX command that prints its arguments, separated by blanks.) This last feature can be expensive, requiring a scan of all sub-directories of */usr/fred*.

There is one exception to the general rules given for patterns. The character *'* at the start of a file name must be explicitly matched.

```
echo *
```

will therefore echo all file names in the current directory not beginning with *'*.

```
echo .*
```

will echo all those file names that begin with *'*. This avoids inadvertent matching of the names *'* and *'..'* which mean 'the current directory' and 'the parent directory' respectively. (Notice that *ls* suppresses information for the files *'* and *'..'*.)

- `ls | grep old | wc -l`
Print the number of files whose name contains the string *old*.
- `cc pgm.c &`
Run *cc* in the background.

2.0 Shell procedures

The shell may be used to read and execute commands contained in a file. For example,

```
sh file [ args ... ]
```

calls the shell to read commands from *file*. Such a file is called a *command procedure* or *shell procedure*. Arguments may be supplied with the call and are referred to in *file* using the positional parameters `$1`, `$2`, For example, if the file *wg* contains

```
who | grep $1
```

then

```
sh wg fred
```

is equivalent to

```
who | grep fred
```

UNIX files have three independent attributes, *read*, *write* and *execute*. The UNIX command *chmod* (1) may be used to make a file executable. For example,

```
chmod +x wg
```

will ensure that the file *wg* has execute status. Following this, the command

```
wg fred
```

is equivalent to

```
sh wg fred
```

This allows shell procedures and programs to be used interchangeably. In either case a new process is created to run the command.

As well as providing names for the positional parameters, the number of positional parameters in the call is available as `$#`. The name of the file being executed is available as `$0`.

A special shell parameter `$*` is used to substitute for all positional parameters except `$0`. A typical use of this is to provide some default arguments, as in,

```
nroff -T450 -ms $*
```

which simply prepends some arguments to those already given.

2.1 Control flow - for

A frequent use of shell procedures is to loop through the arguments (`$1`, `$2`, ...) executing commands once for each argument. An example of such a procedure is *tel* that searches the file `/usr/lib/telnos` that contains lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of *tel* is

match is found the associated *command-list* is executed and execution of the *case* is complete. Since *** is the pattern that matches any string it can be used for the default case.

A word of caution: no check is made to ensure that only one pattern matches the case argument. The first match found defines the set of commands to be executed. In the example below the commands following the second *** will never be executed.

```
case $# in
  *) ... ;;
  *) ... ;;
esac
```

Another example of the use of the *case* construction is to distinguish between different forms of an argument. The following example is a fragment of a *cc* command.

```
for i
do case $i in
  -[ocs]) ... ;;
  -* ) echo 'unknown flag $i' ;;
  *.c) /lib/c0 $i ... ;;
  *)echo 'unexpected argument $i' ;;
esac
done
```

To allow the same commands to be associated with more than one pattern the *case* command provides for alternative patterns separated by a *|*. For example,

```
case $i in
  -x|-y) ...
esac
```

is equivalent to

```
case $i in
  -[xy]) ...
esac
```

The usual quoting conventions apply so that

```
case $i in
  \?) ...
```

will match the character *?*.

2.3 Here documents

The shell procedure *tel* in section 2.1 uses the file */usr/lib/telnetos* to supply the data for *grep*. An alternative is to include this data within the shell procedure as a *here* document, as in,

```
for i
do grep $i <<!
  ...
  fred mh0123
  bert mh0789
  ...
!
done
```

In this example the shell takes the lines between *<<!* and *!* as the standard input for *grep*. The string *!* is arbitrary, the document being terminated by a line that consists of the string following *<<*.

```
echo ${user}
```

which is equivalent to

```
echo $user
```

and is used when the parameter name is followed by a letter or digit. For example,

```
tmp=/tmp/ps
ps a >${tmp}a
```

will direct the output of *ps* to the file */tmp/psa*, whereas,

```
ps a >$tmpa
```

would cause the value of the variable *tmpa* to be substituted.

Except for **\$?** the following are set initially by the shell. **\$?** is set after executing each command.

- \$?** The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully, otherwise a non-zero exit status is returned. Testing the value of return codes is dealt with later under *if* and *while* commands.
- \$#** The number of positional parameters (in decimal). Used, for example, in the *append* command to check the number of parameters.
- \$\$** The process number of this shell (in decimal). Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary file names. For example,


```
ps a >/tmp/ps$$
...
rm /tmp/ps$$
```
- !** The process number of the last process run in the background (in decimal).
- The current shell flags, such as *-x* and *-v*.

Some variables have a special meaning to the shell and should be avoided for general use.

- \$MAIL** When used interactively the shell looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since it was last looked at the shell prints the message *you have mail* before prompting for the next command. This variable is typically set in the file *.profile*, in the user's login directory. For example,

```
MAIL=/usr/spool/mail/fred
```

- \$HOME** The default argument for the *cd* command. The current directory is used to resolve file name references that do not begin with a */*, and is changed using the *cd* command. For example,

```
cd /usr/fred/bin
```

makes the current directory */usr/fred/bin*.

```
cat wn
```

will print on the terminal the file *wn* in this directory. The command *cd* with no argument is equivalent to

```
cd $HOME
```

This variable is also typically set in the the user's login profile.

- \$PATH** A list of directories that contain commands (the *search path*). Each time a command is

```

until test -f file
do sleep 300; done
commands

```

will loop until *file* exists. Each time round the loop it waits for 5 minutes before trying again. (Presumably another process will eventually create the file.)

2.7 Control flow - if

Also available is a general conditional branch of the form.

```

if command-list
then command-list
else command-list
fi

```

that tests the value returned by the last simple command following *if*.

The *if* command may be used in conjunction with the *test* command to test for the existence of a file as in

```

if test -f file
then process file
else do something else
fi

```

An example of the use of *if*, *case* and *for* constructions is given in section 2.10.

A multiple test *if* command of the form

```

if ...
then ...
else if ...
      then ...
      else if ...
            ...
            fi
      fi
fi

```

may be written using an extension of the *if* notation as,

```

if ...
then ...
elif ...
then ...
elif ...
...
fi

```

The following example is the *touch* command which changes the 'last modified' time for a list of files. The command may be used in conjunction with *make* (1) to force recompilation of a list of files.



2.9 Debugging shell procedures

The shell provides two tracing mechanisms to help when debugging shell procedures. The first is invoked within the procedure as

```
set -v
```

(*v* for verbose) and causes lines of the procedure to be printed as they are read. It is useful to help isolate syntax errors. It may be invoked without modifying the procedure by saying

```
sh -v proc ...
```

where *proc* is the name of the shell procedure. This flag may be used in conjunction with the *-n* flag which prevents execution of subsequent commands. (Note that saying *set -n* at a terminal will render the terminal useless until an end-of-file is typed.)

The command

```
set -x
```

will produce an execution trace. Following parameter substitution each command is printed as it is executed. (Try these at the terminal to see what effect they have.) Both flags may be turned off by saying

```
set -
```

and the current setting of the shell flags is available as *\$-*.

2.10 The man command

The following is the *man* command which is used to display sections of the UNIX manual on your terminal. It is called, for example, as

```
man sh
man -t ed
man 2 fork
```

In the first the manual section for *sh* is displayed.. Since no section is specified, section 1 is used. The second example will typeset (*-t* option) the manual section for *ed*. The last prints the *fork* manual page from section 2, which covers system calls.

3.1 Parameter transmission

When a shell procedure is invoked both positional and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a shell procedure by specifying in advance that such parameters are to be exported. For example,

```
export user box
```

marks the variables `user` and `box` for export. When a shell procedure is invoked copies are made of all exportable variables for use within the invoked procedure. Modification of such variables within the procedure does not affect the values in the invoking shell. It is generally true of a shell procedure that it may not modify the state of its caller without explicit request on the part of the caller. (Shared file descriptors are an exception to this rule.)

Names whose value is intended to remain constant may be declared *readonly*. The form of this command is the same as that of the *export* command,

```
readonly name ...
```

Subsequent attempts to set readonly variables are illegal.

3.2 Parameter substitution

If a shell parameter is not set then the null string is substituted for it. For example, if the variable `d` is not set

```
echo $d
```

or

```
echo ${d}
```

will echo nothing. A default string may be given as in

```
echo ${d-}
```

which will echo the value of the variable `d` if it is set and `'` otherwise. The default string is evaluated using the usual quoting conventions so that

```
echo ${d- '*'}
```

will echo `*` if the variable `d` is not set. Similarly

```
echo ${d-$1}
```

will echo the value of `d` if it is set and the value (if any) of `$1` otherwise. A variable may be assigned a default value using the notation

```
echo ${d=}
```

which substitutes the same string as

```
echo ${d-}
```

and if `d` were not previously set then it will be set to the string `'`. (The notation `${...=...}` is not available for positional parameters.)

If there is no sensible default then the notation

```
echo ${d?message}
```

will echo the value of the variable `d` if it has one, otherwise *message* is printed by the shell and execution of the shell procedure is abandoned. If *message* is absent then a standard message is printed. A shell procedure that requires some parameters to be set might start as follows.

```
 : ${user?} ${acct?} ${bin?}
 ...
```

```
echo $X
```

will echo *\$y*.

- blank interpretation

Following the above substitutions the resulting characters are broken into non-blank words (*blank interpretation*). For this purpose 'blanks' are the characters of the string `$IFS`. By default, this string consists of blank, tab and newline. The null string is not regarded as a word unless it is quoted. For example,

```
echo ''
```

will pass on the null string as the first argument to *echo*, whereas

```
echo $null
```

will call *echo* with no arguments if the variable `null` is not set or set to the null string.

- file name generation

Each word is then scanned for the file pattern characters `*`, `?` and `[..]` and an alphabetical list of file names is generated to replace the word. Each such file name is a separate argument.

The evaluations just described also occur in the list of words associated with a `for` loop. Only substitution occurs in the *word* used for a `case` branch.

As well as the quoting mechanisms described earlier using `\` and `'...'` a third quoting mechanism is provided using double quotes. Within double quotes parameter and command substitution occurs but file name generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and may be quoted using `\`.

```
$    parameter substitution
`    command substitution
"    ends the quoted string
\    quotes the special characters $ ` " \
```

For example,

```
echo "$x"
```

will pass the value of the variable `x` as a single argument to *echo*. Similarly,

```
echo "$*"
```

will pass the positional parameters as a single argument and is equivalent to

```
echo "$1 $2 ..."
```

The notation `$@` is the same as `$*` except when it is quoted.

```
echo "$@"
```

will pass the positional parameters, unevaluated, to *echo* and is equivalent to

```
echo "$1" "$2" ...
```

The following table gives, for each quoting mechanism, the shell metacharacters that are evaluated.

- 1 hangup
- 2 interrupt
- 3* quit
- 4* illegal instruction
- 5* trace trap
- 6* IOT instruction
- 7* EMT instruction
- 8* floating point exception
- 9 kill (cannot be caught or ignored)
- 10* bus error
- 11* segmentation violation
- 12* bad argument to system call
- 13 write on a pipe with no one to read it
- 14 alarm clock
- 15 software termination (from *kill* (1))

Figure 3. UNIX signals†

Those signals marked with an asterisk produce a core dump if not caught. However, the shell itself ignores quit which is the only external signal that can cause a dump. The signals in this list of potential interest to shell programs are 1, 2, 3, 14 and 15.

3.6 Fault handling

Shell procedures normally terminate when an interrupt is received from the terminal. The *trap* command is used if some cleaning up is required, such as removing temporary files. For example,

```
trap 'rm /tmp/ps$$; exit' 2
```

sets a trap for signal 2 (terminal interrupt), and if this signal is received will execute the commands

```
rm /tmp/ps$$; exit
```

exit is another built-in command that terminates execution of a shell procedure. The *exit* is required; otherwise, after the trap has been taken, the shell will resume executing the procedure at the place where it was interrupted.

UNIX signals can be handled in one of three ways. They can be ignored, in which case the signal is never sent to the process. They can be caught, in which case the process must decide what action to take when the signal is received. Lastly, they can be left to cause termination of the process without it having to take any further action. If a signal is being ignored on entry to the shell procedure, for example, by invoking it in the background (see 3.7) then *trap* commands (and the signal) are ignored.

The use of *trap* is illustrated by this modified version of the *touch* command (Figure 4). The cleanup action is to remove the file **junk\$\$**.

† Additional signals have been added in Berkeley Unix. See *sigvec(2)* or *signal(3C)* for an up-to-date list.

read x is a built-in command that reads one line from the standard input and places the result in the variable *x*. It returns a non-zero exit status if either an end-of-file is read or an interrupt is received.

3.7 Command execution

To run a command (other than a built-in) the shell first creates a new process using the system call *fork*. The execution environment for the command includes input, output and the states of signals, and is established in the child process before the command is executed. The built-in command *exec* is used in the rare cases when no *fork* is required and simply replaces the shell with a new command. For example, a simple version of the *nohup* command looks like

```
trap `` 1 2 3 15
exec $*
```

The *trap* turns off the signals specified so that they are ignored by subsequently created commands and *exec* replaces the shell by the command specified.

Most forms of input output redirection have already been described. In the following *word* is only subject to parameter and command substitution. No file name generation or blank interpretation takes place so that, for example,

```
echo ...>*.c
```

will write its output into a file whose name is **.c*. Input output specifications are evaluated left to right as they appear in the command.

- > *word* The standard output (file descriptor 1) is sent to the file *word* which is created if it does not already exist.
- >> *word* The standard output is sent to file *word*. If the file exists then output is appended (by seeking to the end); otherwise the file is created.
- < *word* The standard input (file descriptor 0) is taken from the file *word*.
- << *word* The standard input is taken from the lines of shell input that follow up to but not including a line consisting only of *word*. If *word* is quoted then no interpretation of the document occurs. If *word* is not quoted then parameter and command substitution occur and \ is used to quote the characters \ \$ ` and the first character of *word*. In the latter case \newline is ignored (c.f. quoted strings).
- >& *digit* The file descriptor *digit* is duplicated using the system call *dup* (2) and the result is used as the standard output.
- <& *digit* The standard input is duplicated from file descriptor *digit*.
- <&- The standard input is closed.
- >&- The standard output is closed.

Any of the above may be preceded by a digit in which case the file descriptor created is that specified by the digit instead of the default 0 or 1. For example,

```
... 2>file
```

runs a command with message output (file descriptor 2) directed to *file*.

```
... 2>&1
```

runs a command with its standard output and message output merged. (Strictly speaking file descriptor 2 is created by duplicating file descriptor 1 but the effect is usually to merge the two streams.)

The environment for a command run in the background such as

```
list *.c | lpr &
```

is modified in two ways. Firstly, the default standard input for such a command is the empty file */dev/null*. This prevents two processes (the shell and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example,

Appendix A - Grammar

item: *word*
 input-output
 name = value

simple-command: *item*
 simple-command item

command: *simple-command*
 (*command-list*)
 { *command-list* }
 for name do command-list done
 for name in word ... do command-list done
 while command-list do command-list done
 until command-list do command-list done
 case word in case-part ... esac
 if command-list then command-list else-part fi

pipeline: *command*
 pipeline | command

andor: *pipeline*
 andor && pipeline
 andor || pipeline

command-list: *andor*
 command-list ;
 command-list &
 command-list ; andor
 command-list & andor

input-output: *> file*
 < file
 >> word
 << word

file: *word*
 & digit
 & -

case-part: *pattern) command-list ;;*

pattern: *word*
 pattern | word

else-part: **elif command-list then command-list else-part**
 else command-list
 empty

empty:

word: a sequence of non-blank characters

name: a sequence of letters, digits or underscores starting with a letter

digit: **0 1 2 3 4 5 6 7 8 9**

An Introduction to the C shell

William Joy
(revised for 4.3BSD by Mark Seiden)

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

Csh is a new command language interpreter for UNIX† systems. It incorporates good features of other shells and a *history* mechanism similar to the *redo* of INTERLISP. While incorporating many features of other shells which make writing shell programs (shell scripts) easier, most of the features unique to *csh* are designed more for the interactive UNIX user.

UNIX users who have read a general introduction to the system will find a valuable basic explanation of the shell here. Simple terminal interaction with *csh* is possible after reading just the first section of this document. The second section describes the shell's capabilities which you can explore after you have begun to become acquainted with the shell. Later sections introduce features which are useful, but not necessary for all users of the shell.

Additional information includes an appendix listing special characters of the shell and a glossary of terms and commands introduced in this manual.

U
S
D
4

Introduction

A *shell* is a command language interpreter. *Csh* is the name of one particular command interpreter on UNIX. The primary purpose of *csh* is to translate command lines typed at a terminal into system actions, such as invocation of other programs. *Csh* is a user program just like any you might write. Hopefully, *csh* will be a very useful program for you in interacting with the UNIX system.

In addition to this document, you will want to refer to a copy of the UNIX User Reference Manual. The *csh* documentation in section 1 of the manual provides a full description of all features of the shell and is the definitive reference for questions about the shell.

Many words in this document are shown in *italics*. These are important words; names of commands, and words which have special meaning in discussing the shell and UNIX. Many of the words are defined in a glossary at the end of this document. If you don't know what is meant by a word, you should look for it in the glossary.

Acknowledgements

Numerous people have provided good input about previous versions of *csh* and aided in its debugging and in the debugging of its documentation. I would especially like to thank Michael Ubell who made the crucial observation that history commands could be done well over the word structure of input text, and implemented a prototype history mechanism in an older version of the shell. Eric Allman has also provided a large number of useful comments on the shell, helping to unify those

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

1. Terminal usage of the shell

1.1. The basic notion of commands

A *shell* in UNIX acts mostly as a medium through which other *programs* are invoked. While it has a set of *builtin* functions which it performs directly, most commands cause execution of programs that are, in fact, external to the shell. The shell is thus distinguished from the command interpreters of other systems both by the fact that it is just a user program, and by the fact that it is used almost exclusively as a mechanism for invoking other programs.

Commands in the UNIX system consist of a list of strings or *words* interpreted as a *command name* followed by *arguments*. Thus the command

```
mail bill
```

consists of two words. The first word *mail* names the command to be executed, in this case the mail program which sends messages to other users. The shell uses the name of the command in attempting to execute it for you. It will look in a number of *directories* for a file with the name *mail* which is expected to contain the mail program.

The rest of the words of the command are given as *arguments* to the command itself when it is executed. In this case we specified also the argument *bill* which is interpreted by the *mail* program to be the name of a user to whom mail is to be sent. In normal terminal usage we might use the *mail* command as follows.

```
% mail bill
I have a question about the csh documentation.
My document seems to be missing page 5.
Does a page five exist?
      Bill
EOT
%
```

Here we typed a message to send to *bill* and ended this message with a `^D` which sent an end-of-file to the mail program. (Here and throughout this document, the notation “*x*” is to be read “control-*x*” and represents the striking of the *x* key while the control key is held down.) The mail program then echoed the characters ‘EOT’ and transmitted our message. The characters ‘%’ were printed before and after the mail command by the shell to indicate that input was needed.

After typing the ‘%’ prompt the shell was reading command input from our terminal. We typed a complete command ‘mail bill’. The shell then executed the *mail* program with argument *bill* and went dormant waiting for it to complete. The mail program then read input from our terminal until we signalled an end-of-file via typing a `^D` after which the shell noticed that mail had completed and signalled us that it was ready to read from the terminal again by printing another ‘%’ prompt.

This is the essential pattern of all interaction with UNIX through the shell. A complete command is typed at the terminal, the shell executes the command and when this execution completes, it prompts for a new command. If you run the editor for an hour, the shell will patiently wait for you to finish editing and obediently prompt you again whenever you finish editing.

An example of a useful command you can execute now is the *tset* command, which sets the default *erase* and *kill* characters on your terminal – the erase character erases the last character you typed and the kill character erases the entire line you have entered so far. By default, the erase character is the delete key (equivalent to “`?`”) and the kill character is “`U`”. Some people prefer to make the erase character the backspace key (equivalent to “`H`”). You can make this be true by typing

```
tset -e
```

which tells the program *tset* to set the erase character to *tset*’s default setting for this character (a backspace).

1.4. Metacharacters in the shell

The shell has a large number of special characters (like '>') which indicate special functions. We say that these notations have *syntactic* and *semantic* meaning to the shell. In general, most characters which are neither letters nor digits have special meaning to the shell. We shall shortly learn a means of *quotation* which allows us to use *metacharacters* without the shell treating them in any special way.

Metacharacters normally have effect only when the shell is reading our input. We need not worry about placing shell metacharacters in a letter we are sending via *mail*, or when we are typing in text or data to some other program. Note that the shell is only reading input when it has prompted with '%' (although we can type our input even before it prompts).

1.5. Input from files; pipelines

We learned above how to *redirect* the *standard output* of a command to a file. It is also possible to redirect the *standard input* of a command from a file. This is not often necessary since most commands will read from a file whose name is given as an argument. We can give the command

```
sort < data
```

to run the *sort* command with standard input, where the command normally reads its input, from the file 'data'. We would more likely say

```
sort data
```

letting the *sort* command open the file 'data' for input itself since this is less to type.

We should note that if we just typed

```
sort
```

then the *sort* program would sort lines from its *standard input*. Since we did not *redirect* the standard input, it would sort lines as we typed them on the terminal until we typed a ^D to indicate an end-of-file.

A most useful capability is the ability to combine the standard output of one command with the standard input of another, i.e. to run the commands in a sequence known as a *pipeline*. For instance the command

```
ls -s
```

normally produces a list of the files in our directory with the size of each in blocks of 512 characters. If we are interested in learning which of our files is largest we may wish to have this sorted by size rather than by name, which is the default way in which *ls* sorts. We could look at the many options of *ls* to see if there was an option to do this but would eventually discover that there is not. Instead we can use a couple of simple options of the *sort* command, combining it with *ls* to get what we want.

The *-n* option of *sort* specifies a numeric sort rather than an alphabetic sort. Thus

```
ls -s | sort -n
```

specifies that the output of the *ls* command run with the option *-s* is to be *piped* to the command *sort* run with the numeric sort option. This would give us a sorted list of our files by size, but with the smallest first. We could then use the *-r* reverse sort option and the *head* command in combination with the previous command doing

```
ls -s | sort -n -r | head -5
```

Here we have taken a list of our files sorted alphabetically, each with the size in blocks. We have run this to the standard input of the *sort* command asking it to sort numerically in reverse order (largest first). This output has then been run into the command *head* which gives us the first few lines. In this case we have asked *head* for the first 5 lines. Thus this command gives us the names and sizes of our 5 largest files.




```
echo ? ?? ???
```

will echo a line of filenames; first those with one character names, then those with two character names, and finally those with three character names. The names of each length will be independently sorted.

Another mechanism consists of a sequence of characters between '[' and ']'. This metasequence matches any single character from the enclosed set. Thus

```
prog.[co]
```

will match

```
prog.c prog.o
```

in the example above. We can also place two characters around a '-' in this notation to denote a range. Thus

```
chap.[1-5]
```

might match files

```
chap.1 chap.2 chap.3 chap.4 chap.5
```

if they existed. This is shorthand for

```
chap.[12345]
```

and otherwise equivalent.

An important point to note is that if a list of argument words to a command (an *argument list*) contains filename expansion syntax, and if this filename expansion syntax fails to match any existing file names, then the shell considers this to be an error and prints a diagnostic

```
No match.
```

and does not execute the command.

Another very important point is that files with the character '.' at the beginning are treated specially. Neither '*' or '?' or the '[' ']' mechanism will match it. This prevents accidental matching of the filenames '.' and '..' in the working directory which have special meaning to the system, as well as other files such as *.cshrc* which are not normally visible. We will discuss the special role of the file *.cshrc* later.

Another filename expansion mechanism gives access to the pathname of the *home* directory of other users. This notation consists of the character '~' (tilde) followed by another user's login name. For instance the word '~bill' would map to the pathname '/usr/bill' if the home directory for 'bill' was '/usr/bill'. Since, on large systems, users may have login directories scattered over many different disk volumes with different prefix directory names, this notation provides a convenient way of accessing the files of other users.

A special case of this notation consists of a '~' alone, e.g. '~/mbox'. This notation is expanded by the shell into the file 'mbox' in your *home* directory, i.e. into '/usr/bill/mbox' for me on Ernie Co-vax, the UCB Computer Science Department VAX machine, where this document was prepared. This can be very useful if you have used *cd* to change to another directory and have found a file you wish to copy using *cp*. If I give the command

```
cp thatfile ~
```

the shell will expand this command to

```
cp thatfile /usr/bill
```

since my home directory is /usr/bill.

There also exists a mechanism using the characters '{' and '}' for abbreviating a set of words which have common parts but cannot be abbreviated by the above mechanisms because they are not

If a command has its standard input redirected from a file, then it will normally terminate when it reaches the end of this file. Thus if we execute

```
mail bill < prepared.text
```

the mail command will terminate without our typing a `^D`. This is because it read to the end-of-file of our file 'prepared.text' in which we placed a message for 'bill' with an editor program. We could also have done

```
cat prepared.text | mail bill
```

since the *cat* command would then have written the text through the pipe to the standard input of the mail command. When the *cat* command completed it would have terminated, closing down the pipeline and the *mail* command would have received an end-of-file from it and terminated. Using a pipe here is more complicated than redirecting input so we would more likely use the first form. These commands could also have been stopped by sending an INTERRUPT.

Another possibility for stopping a command is to suspend its execution temporarily, with the possibility of continuing execution later. This is done by sending a STOP signal via typing a `^Z`. This signal causes all commands running on the terminal (usually one but more if a pipeline is executing) to become suspended. The shell notices that the command(s) have been suspended, types 'Stopped' and then prompts for a new command. The previously executing command has been suspended, but otherwise unaffected by the STOP signal. Any other commands can be executed while the original command remains suspended. The suspended command can be continued using the *fg* command with no arguments. The shell will then retype the command to remind you which command is being continued, and cause the command to resume execution. Unless any input files in use by the suspended command have been changed in the meantime, the suspension has no effect whatsoever on the execution of the command. This feature can be very useful during editing, when you need to look at another file before continuing. An example of command suspension follows.

```
% mail harold
Someone just copied a big file into my directory and its name is
^Z
Stopped
% ls
funnyfile
prog.c
prog.o
% jobs
[1] + Stopped          mail harold
% fg
mail harold
funnyfile. Do you know who did it?
EOT
%
```

In this example someone was sending a message to Harold and forgot the name of the file he wanted to mention. The mail command was suspended by typing `^Z`. When the shell noticed that the mail program was suspended, it typed 'Stopped' and prompted for a new command. Then the *ls* command was typed to find out the name of the file. The *jobs* command was run to find out which command was suspended. At this time the *fg* command was typed to continue execution of the mail program. Input to the mail program was then continued and ended with a `^D` which indicated the end of the message at which time the mail program typed EOT. The *jobs* command will show which commands are suspended. The `^Z` should only be typed at the beginning of a line since everything typed on the current line is discarded when a signal is sent from the keyboard. This also happens on INTERRUPT, and QUIT signals. More information on suspending jobs and controlling them is given in section 2.6.

2. Details on the shell for terminal users

2.1. Shell startup and termination

When you login, the shell is started by the system in your *home* directory and begins by reading commands from a file *.cshrc* in this directory. All shells which you may start during your terminal session will read from this file. We will later see what kinds of commands are usefully placed there. For now we need not have this file and the shell does not complain about its absence.

A *login shell*, executed after you login to the system, will, after it reads commands from *.cshrc*, read commands from a file *.login* also in your home directory. This file contains commands which you wish to do each time you login to the UNIX system. My *.login* file looks something like:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
echo "${prompt}users" ; users
alias ts \
    'set noglob ; eval `tset -s -m dialup:c100rv4pna -m plugboard:?hp2621nl *`';
ts; stty intr ^C kill ^U crt
set time=15 history=10
msgs -f
if (-e $mail) then
    echo "${prompt}mail"
    mail
endif
```

This file contains several commands to be executed by UNIX each time I login. The first is a *set* command which is interpreted directly by the shell. It sets the shell variable *ignoreeof* which causes the shell to not log me off if I hit ^D. Rather, I use the *logout* command to log off of the system. By setting the *mail* variable, I ask the shell to watch for incoming mail to me. Every 5 minutes the shell looks for this file and tells me if more mail has arrived there. An alternative to this is to put the command

```
biff y
```

in place of this *set*; this will cause me to be notified immediately when mail arrives, and to be shown the first few lines of the new message.

Next I set the shell variable 'time' to '15' causing the shell to automatically print out statistics lines for commands which execute for at least 15 seconds of CPU time. The variable 'history' is set to 10 indicating that I want the shell to remember the last 10 commands I type in its *history list*, (described later).

I create an *alias* "ts" which executes a *tset(1)* command setting up the modes of the terminal. The parameters to *tset* indicate the kinds of terminal which I usually use when not on a hardwired port. I then execute "ts" and also use the *stty* command to change the interrupt character to ^C and the line kill character to ^U.

I then run the 'msgs' program, which provides me with any system messages which I have not seen before; the '-f' option here prevents it from telling me anything if there are no new messages. Finally, if my mailbox file exists, then I run the 'mail' program to process my mail.

When the 'mail' and 'msgs' programs finish, the shell will finish processing my *.login* file and begin reading commands from the terminal, prompting for each with '%'. When I log off (by giving the *logout* command) the shell will print 'logout' and execute commands from the file '.logout' if it exists in my home directory. After that the shell will terminate and UNIX will log me off the system. If the system is not going down, I will receive a new login message. In any case, after the 'logout' message the shell is committed to terminating and will take no further input from my terminal.

Other useful built in variables are the variable *home* which shows your home directory, *cwd* which contains your current working directory, the variable *ignoreeof* which can be set in your *.login* file to tell the shell not to exit when it receives an end-of-file from a terminal (as described above). The variable 'ignoreeof' is one of several variables which the shell does not care about the value of, only whether they are *set* or *unset*. Thus to set this variable you simply do

```
set ignoreeof
```

and to unset it do

```
unset ignoreeof
```

These give the variable 'ignoreeof' no value, but none is desired or required.

Finally, some other built-in shell variables of use are the variables *noclobber* and *mail*. The metasyntax

```
> filename
```

which redirects the standard output of a command will overwrite and destroy the previous contents of the named file. In this way you may accidentally overwrite a file which is valuable. If you would prefer that the shell not overwrite files in this way you can

```
set noclobber
```

in your *.login* file. Then trying to do

```
date > now
```

would cause a diagnostic if 'now' existed already. You could type

```
date >! now
```

if you really wanted to overwrite the contents of 'now'. The '>!' is a special metasyntax indicating that clobbering the file is ok.†

2.3. The shell's history list

The shell can maintain a *history list* into which it places the words of previous commands. It is possible to use a notation to reuse commands or words from commands in forming new commands. This mechanism can be used to repeat previous commands or to correct minor typing mistakes in commands.

The following figure gives a sample session involving typical usage of the history mechanism of the shell. In this example we have a very simple C program which has a bug (or two) in it in the file 'bug.c', which we 'cat' out on our terminal. We then try to run the C compiler on it, referring to the file again as '\$', meaning the last argument to the previous command. Here the '!' is the history mechanism invocation metacharacter, and the '\$' stands for the last argument, by analogy to '\$' in the editor which stands for the end of the line. The shell echoed the command, as it would have been typed without use of the history mechanism, and then executed it. The compilation yielded error diagnostics so we now run the editor on the file we were trying to compile, fix the bug, and run the C compiler again, this time referring to this command simply as '!c', which repeats the last command which started with the letter 'c'. If there were other commands starting with 'c' done recently we could have said '!cc' or even '!cc:p' which would have printed the last command starting with 'cc' without executing it.

After this recompilation, we ran the resulting 'a.out' file, and then noting that there still was a bug, ran the editor again. After fixing the program we ran the C compiler again, but tacked onto the command an extra '-o bug' telling the compiler to place the resultant binary in the file 'bug' rather than 'a.out'. In general, the history mechanisms may be used anywhere in the formation of new

†The space between the '!' and the word 'now' is critical here, as '!now' would be an invocation of the *history* mechanism, and have a totally different effect.

commands and other characters may be placed before and after the substituted commands.

We then ran the 'size' command to see how large the binary program images we have created were, and then an 'ls -l' command with the same argument list, denoting the argument list '*'. Finally we ran the program 'bug' to see that its output is indeed correct.

To make a numbered listing of the program we ran the 'num' command on the file 'bug.c'. In order to compress out blank lines in the output of 'num' we ran the output through the filter 'ssp', but misspelled it as spp. To correct this we used a shell substitute, placing the old text and new text between '' characters. This is similar to the substitute command in the editor. Finally, we repeated the same command with '!!', but sent its output to the line printer.

There are other mechanisms available for repeating commands. The *history* command prints out a number of previous commands with numbers by which they can be referenced. There is a way to refer to a previous command by searching for a string which appeared in it, and there are other, less useful, ways to select arguments to include in a new command. A complete description of all these mechanisms is given in the C shell manual pages in the UNIX Programmer's Manual.

2.4. Aliases

The shell has an *alias* mechanism which can be used to make transformations on input commands. This mechanism can be used to simplify the commands you type, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to a macro facility. Some of the features obtained by aliasing can be obtained also using shell command files, but these take place in another instance of the shell and cannot directly affect the current shells environment or involve commands such as *cd* which must be done in the current shell.

As an example, suppose that there is a new version of the mail program on the system called 'newmail' you wish to use, rather than the standard mail program which is called 'mail'. If you place the shell command

```
alias mail newmail
```

in your *.cshrc* file, the shell will transform an input line of the form

```
mail bill
```

into a call on 'newmail'. More generally, suppose we wish the command 'ls' to always show sizes of files, that is to always do '-s'. We can do

```
alias ls ls -s
```

or even

```
alias dir ls -s
```

creating a new command syntax 'dir' which does an 'ls -s'. If we say

```
dir ~bill
```

then the shell will translate this to

```
ls -s /mnt/bill
```

Thus the *alias* mechanism can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to define aliases which contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism. Thus the definition

```
alias cd 'cd \!* ; ls '
```

would do an *ls* command after each change directory *cd* command. We enclosed the entire alias definition in '' characters to prevent most substitutions from occurring and the character ';' from

is ready for another command. The job runs *in the background* at the same time that normal jobs, called *foreground* jobs, continue to be read and executed by the shell one at a time. Thus

```
du > usage &
```

would run the *du* program, which reports on the disk usage of your working directory (as well as any directories below it), put the output into the file 'usage' and return immediately with a prompt for the next command without waiting for *du* to finish. The *du* program would continue executing in the background until it finished, even though you can type and execute more commands in the mean time. When a background job terminates, a message is typed by the shell just before the next prompt telling you that the job has completed. In the following example the *du* job finishes sometime during the execution of the *mail* command and its completion is reported just before the prompt after the *mail* job is finished.

```
% du > usage &
[1] 503
% mail bill
How do you know when a background job is finished?
EOT
[1] - Done          du > usage
%
```

If the job did not terminate normally the 'Done' message might say something else like 'Killed'. If you want the terminations of background jobs to be reported at the time they occur (possibly interrupting the output of other foreground jobs), you can set the *notify* variable. In the previous example this would mean that the 'Done' message might have come right in the middle of the message to Bill. Background jobs are unaffected by any signals from the keyboard like the STOP, INTERRUPT, or QUIT signals mentioned earlier.

Jobs are recorded in a table inside the shell until they terminate. In this table, the shell remembers the command names, arguments and the *process numbers* of all commands in the job as well as the working directory where the job was started. Each job in the table is either running *in the foreground* with the shell waiting for it to terminate, running *in the background*, or *suspended*. Only one job can be running in the foreground at one time, but several jobs can be suspended or running in the background at once. As each job is started, it is assigned a small identifying number called the *job number* which can be used later to refer to the job in the commands described below. Job numbers remain the same until the job terminates and then are re-used.

When a job is started in the background using '&', its number, as well as the process numbers of all its (top level) commands, is typed by the shell before prompting you for another command. For example,

```
% ls -s | sort -n > usage &
[2] 2034 2035
%
```

runs the 'ls' program with the '-s' options, pipes this output into the 'sort' program with the '-n' option which puts its output into the file 'usage'. Since the '&' was at the end of the line, these two programs were started together as a background job. After starting the job, the shell prints the job number in brackets (2 in this case) followed by the process number of each program started in the job. Then the shell immediately prompts for a new command, leaving the job running simultaneously.

As mentioned in section 1.8, foreground jobs become *suspended* by typing ^Z which sends a STOP signal to the currently running foreground job. A background job can become suspended by using the *stop* command described below. When jobs are suspended they merely stop any further progress until started again, either in the foreground or the background. The shell notices when a job becomes stopped and reports this fact, much like it reports the termination of background jobs. For foreground jobs this looks like



```

% du > usage &
[1] 3398
% ls -s | sort -n > myfile &
[2] 3405
% mail bill
^Z
Stopped
% jobs
[1] - Running          du > usage
[2]  Running          ls -s | sort -n > myfile
[3] + Stopped         mail bill
% fg %ls
ls -s | sort -n > myfile
% more myfile

```

The *fg* command runs a suspended or background job in the foreground. It is used to restart a previously suspended job or change a background job to run in the foreground (allowing signals or input from the terminal). In the above example we used *fg* to change the 'ls' job from the background to the foreground since we wanted to wait for it to finish before looking at its output file. The *bg* command runs a suspended job in the background. It is usually used after stopping the currently running foreground job with the STOP signal. The combination of the STOP signal and the *bg* command changes a foreground job into a background job. The *stop* command suspends a background job.

The *kill* command terminates a background or suspended job immediately. In addition to jobs, it may be given process numbers as arguments, as printed by *ps*. Thus, in the example above, the running *du* command could have been terminated by the command

```

% kill %1
[1] Terminated      du > usage
%

```

The *notify* command (not the variable mentioned earlier) indicates that the termination of a specific job should be reported at the time it finishes instead of waiting for the next prompt.

If a job running in the background tries to read input from the terminal it is automatically stopped. When such a job is then run in the foreground, input can be given to the job. If desired, the job can be run in the background again until it requests input again. This is illustrated in the following sequence where the 's' command in the text editor might take a long time.

```

% ed bigfile
120000
1,$s/thisword/thatword/
^Z
Stopped
% bg
[1] ed bigfile &
%
... some foreground commands
[1] Stopped (tty input)  ed bigfile
% fg
ed bigfile
w
120000
q
%

```

```
% pwd
/usr/bill
% mkdir newspaper
% chdir newspaper
% pwd
/usr/bill/newspaper
%
```

the user has created and moved to the directory *newspaper*. where, for example, he might place a group of related files.

No matter where you have moved to in a directory hierarchy, you can return to your 'home' login directory by doing just

```
cd
```

with no arguments. The name '.' always means the directory above the current one in the hierarchy, thus

```
cd ..
```

changes the shell's working directory to the one directly above the current one. The name '..' can be used in any pathname, thus,

```
cd ../programs
```

means change to the directory 'programs' contained in the directory above the current one. If you have several directories for different projects under, say, your home directory, this shorthand notation permits you to switch easily between them.

The shell always remembers the pathname of its current working directory in the variable *cwd*. The shell can also be requested to remember the previous directory when you change to a new working directory. If the 'push directory' command *pushd* is used in place of the *cd* command, the shell saves the name of the current working directory on a *directory stack* before changing to the new one. You can see this list at any time by typing the 'directories' command *dirs*.

```
% pushd newspaper/references
~/newspaper/references ~
% pushd /usr/lib/tmac
/usr/lib/tmac ~/newspaper/references ~
% dirs
/usr/lib/tmac ~/newspaper/references ~
% popd
~/newspaper/references ~
% popd
~
%
```

The list is printed in a horizontal line, reading left to right, with a tilde (~) as shorthand for your home directory—in this case '/usr/bill'. The directory stack is printed whenever there is more than one entry on it and it changes. It is also printed by a *dirs* command. *Dirs* is usually faster and more informative than *pwd* since it shows the current working directory as well as any other directories remembered in the stack.

The *pushd* command with no argument alternates the current directory with the first directory in the list. The 'pop directory' *popd* command without an argument returns you to the directory you were in prior to the current one, discarding the previous current directory from the stack (forgetting it). Typing *popd* several times in a series takes you backward through the directories you had been in (changed to) by *pushd* command. There are other options to *pushd* and *popd* to manipulate the contents of the directory stack and to change to directories not at the top of the stack; see the *cs*h manual page for details.

list. You can use this number to refer to this command in a history substitution. Thus you could
 set `prompt= '\! % '`

Note that the ‘!’ character had to be *escaped* here even within ‘’’ characters.

The *limit* command is used to restrict use of resources. With no arguments it prints the current limitations:

```

cputime      unlimited
filesize     unlimited
datasize     5616 kbytes
stacksize    512 kbytes
coredumpsize unlimited
  
```

Limits can be set, e.g.:

```
limit coredumpsize 128k
```

Most reasonable units abbreviations will work; see the *cs* manual page for more details.

The *logout* command can be used to terminate a login shell which has *ignoreeof* set.

The *rehash* command causes the shell to recompute a table of where commands are located. This is necessary if you add a command to a directory in the current shell’s search path and wish the shell to find it, since otherwise the hashing algorithm may tell the shell that the command wasn’t in that directory when the hash table was computed.

The *repeat* command can be used to repeat a command several times. Thus to make 5 copies of the file *one* in the file *five* you could do

```
repeat 5 cat one >> five
```

The *setenv* command can be used to set variables in the environment. Thus

```
setenv TERM adm3a
```

will set the value of the environment variable *TERM* to ‘adm3a’. A user program *printenv* exists which will print out the environment. It might then show:

```

% printenv
HOME=/usr/bill
SHELL=/bin/csh
PATH=:/usr/ucb:/bin:/usr/bin:/usr/local
TERM=adm3a
USER=bill
%
  
```

The *source* command can be used to force the current shell to read commands from a file. Thus

```
source .cshrc
```

can be used after editing in a change to the *.cshrc* file which you wish to take effect right away.

The *time* command can be used to cause a command to be timed no matter how much CPU time it takes. Thus

3. Shell control structures and command scripts

3.1. Introduction

It is possible to place commands in files and to cause shells to be invoked to read and execute commands from these files, which are called *shell scripts*. We here detail those features of the shell useful to the writers of such scripts.

3.2. Make

It is important to first note what shell scripts are *not* useful for. There is a program called *make* which is very useful for maintaining a group of related files or performing sets of operations on related files. For instance a large program consisting of one or more files can have its dependencies described in a *makefile* which contains definitions of the commands used to create these different files when changes occur. Definitions of the means for printing listings, cleaning up the directory in which the files reside, and installing the resultant programs are easily, and most appropriately placed in this *makefile*. This format is superior and preferable to maintaining a group of shell procedures to maintain these files.

Similarly when working on a document a *makefile* may be created which defines how different versions of the document are to be created and which options of *nroff* or *troff* are appropriate.

3.3. Invocation and the argv variable

A *cs*h command script may be interpreted by saying

```
% csh script ...
```

where *script* is the name of the file containing a group of *cs*h commands and ‘...’ is replaced by a sequence of arguments. The shell places these arguments in the variable *argv* and then begins to read commands from the script. These parameters are then available through the same mechanisms which are used to reference any other shell variables.

If you make the file ‘script’ executable by doing

```
chmod 755 script
```

and place a shell comment at the beginning of the shell script (i.e. begin the file with a ‘#’ character) then a ‘/bin/csh’ will automatically be invoked to execute ‘script’ when you type

```
script
```

If the file does not begin with a ‘#’ then the standard shell ‘/bin/sh’ will be used to execute it. This allows you to convert your older shell scripts to use *cs*h at your convenience.

3.4. Variable substitution

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism know as *variable substitution* is done on these words. Keyed by the character ‘\$’ this substitution replaces the names of variables by their values. Thus

```
echo $argv
```

when placed in a command script would cause the current value of the variable *argv* to be echoed to the output of the shell script. It is an error for *argv* to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation

```
 $?name
```

expands to ‘1’ if name is *set* or to ‘0’ if name is not *set*. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to

One minor difference between '\$n' and '\$argv[n]' should be noted here. The form '\$argv[n]' will yield an error if *n* is not in the range '1- \$#argv' while '\$n' will never yield an out of range subscript error. This is for compatibility with the way older shells handled parameters.

Another important point is that it is never an error to give a subrange of the form 'n-'; if there are less than *n* components of the given variable then no words are substituted. A range of the form 'm-n' likewise returns an empty vector without giving an error when *m* exceeds the number of elements of the given variable, provided the subscript *n* is in range.

3.5. Expressions

In order for interesting shell scripts to be constructed it must be possible to evaluate expressions in the shell based on the values of variables. In fact, all the arithmetic operations of the language C are available in the shell with the same precedence that they have in C. In particular, the operations '=' and '!=' compare strings and the operators '&&' and '|' implement the boolean and/or operations. The special operators '=~' and '!~' are similar to '=' and '!=' except that the string on the right side can have pattern matching characters (like *, ? or []) and the test is whether the string on the left matches the pattern on the right.

The shell also allows file enquiries of the form

```
-? filename
```

where '?' is replace by a number of single characters. For instance the expression primitive

```
-e filename
```

tell whether the file 'filename' exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or has non-zero length.

It is possible to test whether a command terminates normally, by a primitive of the form

```
{ command }
```

which returns true, i.e. '1' if the command succeeds exiting normally with exit status 0, or '0' if the command terminates abnormally or with exit status non-zero. If more detailed information about the execution status of a command is required, it can be executed and the variable '\$status' examined in the next command. Since '\$status' is set by every command, it is very transient. It can be saved if it is inconvenient to use it only in the single immediately following command.

For a full list of expression components available see the manual section for the shell.

3.6. Sample shell script

A sample shell script which makes use of the expression mechanism of the shell and some of its control structure follows:

The shell does have another form of the if statement of the form

```
if ( expression ) command
```

which can be written

```
if ( expression ) \  
    command
```

Here we have escaped the newline for the sake of appearance. The command must not involve '|', '&' or ';' and must not be another control command. The second form requires the final '\ ' immediately precede the end-of-line.

The more general *if* statements above also admit a sequence of *else-if* pairs followed by a single *else* and an *endif*, e.g.:

```
if ( expression ) then  
    commands  
else if (expression ) then  
    commands  
...  
else  
    commands  
endif
```

Another important mechanism used in shell scripts is the ':' modifier. We can use the modifier 'r' here to extract a root of a filename or 'e' to extract the *extension*. Thus if the variable *i* has the value '/mnt/foo.bar' then

```
% echo $i $i:r $i:e  
/mnt/foo.bar /mnt/foo bar  
%
```

shows how the 'r' modifier strips off the trailing '.bar' and the the 'e' modifier leaves only the 'bar'. Other modifiers will take off the last component of a pathname leaving the head 'h' or all but the last component of a pathname leaving the tail 't'. These modifiers are fully described in the *cs* manual pages in the User's Reference Manual. It is also possible to use the *command substitution* mechanism described in the next major section to perform modifications on strings to then reenter the shell's environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the ':' modification mechanism.‡ Finally, we note that the character '#' lexically introduces a shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a '#' are discarded by the shell. This character can be quoted using '' or '\ ' to place it in an argument word.

3.7. Other control structures

The shell also has control structures *while* and *switch* similar to those of C. These take the forms

‡ It is also important to note that the current implementation of the shell limits the number of ':' modifiers on a '\$' substitution to 1. Thus

```
% echo $i $i:h:t  
/a/b/c /a/b:t  
%
```

does not do what one would expect.

intervening lines. In general, if any part of the word following the '<<' which the shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form '1,\$' in our editor script we needed to insure that this '\$' was not variable substituted. We could also have insured this by preceding the '\$' here with a '\', i.e.:

```
1,\$s/[ ]*//
```

but quoting the 'EOF' terminator is a more reliable way of achieving the same thing.

3.9. Catching interrupts

If our shell script creates temporary files, we may wish to catch interruptions of the shell script so that we can clean up these files. We can then do

```
onintr label
```

where *label* is a label in our program. If an interrupt is received the shell will do a 'goto label' and we can remove the temporary files and then do an *exit* command (which is built in to the shell) to exit from the shell script. If we wish to exit with a non-zero status we can do

```
exit(1)
```

e.g. to exit with status '1'.

3.10. What else?

There are other features of the shell useful to writers of shell procedures. The *verbose* and *echo* options and the related *-v* and *-x* command line options can be used to help trace the actions of the shell. The *-n* option causes the shell only to read commands and not to execute them and may sometimes be of use.

One other thing to note is that *csh* will not execute shell scripts which do not begin with the character '#', that is shell scripts that do not begin with a comment. Similarly, the '/bin/sh' on your system may well defer to 'csh' to interpret shell scripts which begin with '#'. This allows shell scripts for both shells to live in harmony.

There is also another quotation mechanism using "" which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as '' does.



Astr1B Astr2B ... AstrnB

This expansion occurs before the other filename expansions, and may be applied recursively (i.e. nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments which are not filenames, but which have common parts.

A typical use of this would be

```
mkdir ~/{hdrs,retrofit,csh}
```

to make subdirectories 'hdrs', 'retrofit' and 'csh' in your home directory. This mechanism is most useful when the common prefix is longer than in this example, i.e.

```
chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

4.3. Command substitution

A command enclosed in `` characters is replaced, just before filenames are expanded, by the output from that command. Thus it is possible to do

```
set pwd=`pwd`
```

to save the current directory in the variable *pwd* or to do

```
ex `grep -l TRACE *.c`
```

to run the editor *ex* supplying as arguments those files whose names end in '.c' which have the string 'TRACE' in them.*

4.4. Other details not covered here

In particular circumstances it may be necessary to know the exact nature and order of different substitutions performed by the shell. The exact meaning of certain combinations of quotations is also occasionally important. These are detailed fully in its manual section.

The shell has a number of command line option flags mostly of use in writing UNIX programs, and debugging shell scripts. See the *csh(1)* manual section for a list of these options.

*Command expansion also occurs in input redirected with '<<' and within '' quotations. Refer to the shell manual section for full details.

Glossary

This glossary lists the most important terms introduced in the introduction to the shell and gives references to sections of the shell document for further information about them. References of the form 'pr (1)' indicate that the command *pr* is in the UNIX User Reference manual in section 1. You can look at an online copy of its manual page by doing

man 1 pr

References of the form (2.5) indicate that more information can be found in section 2.5 of this manual.

. Your current directory has the name '.' as well as the name printed by the command *pwd*; see also *dirs*. The current directory '.' is usually the first *component* of the search path contained in the variable *path*, thus commands which are in '.' are found first (2.2). The character '.' is also used in separating *components* of filenames (1.6). The character '.' at the beginning of a *component* of a *pathname* is treated specially and not matched by the *filename expansion* metacharacters '?', '*', and '[' ']' pairs (1.6).

.. Each directory has a file '..' in it which is a reference to its parent directory. After changing into the directory with *chdir*, i.e.

chdir paper

you can return to the parent directory by doing

chdir ..

The current directory is printed by *pwd* (2.7).

a.out Compilers which create executable images create them, by default, in the file *a.out*. for historical reasons (2.3).

absolute pathname

A *pathname* which begins with a '/' is *absolute* since it specifies the *path* of directories from the beginning of the entire directory system - called the *root* directory. *Pathnames* which are not *absolute* are called *relative* (see definition of *relative pathname*) (1.6).

alias An *alias* specifies a shorter or different name for a UNIX command, or a transformation on a command to be performed in the shell. The shell has a command *alias* which establishes *aliases* and can print their current values. The command *unalias* is used to remove *aliases* (2.4).

argument Commands in UNIX receive a list of *argument* words. Thus the command

echo a b c

consists of the *command name* 'echo' and three *argument* words 'a', 'b' and 'c'. The set of *arguments* after the *command name* is said to be the *argument list* of the command (1.1).

argv The list of arguments to a command written in the shell language (a shell script or shell procedure) is stored in a variable called *argv* within the shell. This name is taken from the conventional name in the C programming language (3.4).

background Commands started without waiting for them to complete are called *background* commands (2.6).

base A filename is sometimes thought of as consisting of a *base* part, before any '.' character, and an *extension* - the part after the '.'. See *filename* and *extension* (1.6) and *basename* (1).

bg The *bg* command causes a *suspended* job to continue execution in the *background* (2.6).



- continue** A builtin command which causes execution of the enclosing *foreach* or *while* loop to cycle prematurely. Similar to the *continue* command in the programming language C (3.6).
- control-** Certain special characters, called *control* characters, are produced by holding down the CONTROL key on your terminal and simultaneously pressing another character, much like the SHIFT key is used to produce upper case characters. Thus *control-c* is produced by holding down the CONTROL key while pressing the 'c' key. Usually UNIX prints an caret (^) followed by the corresponding letter when you type a *control* character (e.g. "C" for *control-c* (1.8)).
- core dump** When a program terminates abnormally, the system places an image of its current state in a file named 'core'. This *core dump* can be examined with the system debugger 'adb (1)' or 'sdb (1)' in order to determine what went wrong with the program (1.8). If the shell produces a message of the form
- Illegal instruction (core dumped)
- (where 'Illegal instruction' is only one of several possible messages), you should report this to the author of the program or a system administrator, saving the 'core' file.
- cp** The *cp* (copy) program is used to copy the contents of one file into another file. It is one of the most commonly used UNIX commands (1.6).
- csh** The name of the shell program that this document describes.
- .cshrc** The file *.cshrc* in your *home* directory is read by each shell as it begins execution. It is usually used to change the setting of the variable *path* and to set *alias* parameters which are to take effect globally (2.1).
- cwd** The *cwd* variable in the shell holds the *absolute pathname* of the current *working directory*. It is changed by the shell whenever your current *working directory* changes and should not be changed otherwise (2.2).
- date** The *date* command prints the current date and time (1.3).
- debugging** *Debugging* is the process of correcting mistakes in programs and shell scripts. The shell has several options and variables which may be used to aid in shell *debugging* (4.4).
- default:** The label *default:* is used within shell *switch* statements, as it is in the C language to label the code to be executed if none of the *case* labels matches the value switched on (3.7).
- DELETE** The DELETE or RUBOUT key on the terminal normally causes an interrupt to be sent to the current job. Many users change the interrupt character to be ^C.
- detached** A command that continues running in the *background* after you logout is said to be *detached*.
- diagnostic** An error message produced by a program is often referred to as a *diagnostic*. Most error messages are not written to the *standard output*, since that is often directed away from the terminal (1.3, 1.5). Error messages are instead written to the *diagnostic output* which may be directed away from the terminal, but usually is not. Thus *diagnostics* will usually appear on the terminal (2.5).
- directory** A structure which contains files. At any time you are in one particular *directory* whose names can be printed by the command *pwd*. The *chdir* command will change you to another *directory*, and make the files in that *directory* visible. The *directory* in which you are when you first login is your *home* directory (1.1, 2.7).
- directory stack** The shell saves the names of previous *working directories* in the *directory stack* when you change your current *working directory* via the *pushd* command. The *directory stack* can be printed by using the *dirs* command, which includes your current

'prog.c' were a C program, then the object file for this program would be stored in 'prog.o'. Similarly a paper written with the '-me' nroff macro package might be stored in 'paper.me' while a formatted version of this paper might be kept in 'paper.out' and a list of spelling errors in 'paper.errs' (1.6).

fg The *job control* command *fg* is used to run a *background* or *suspended* job in the *foreground* (1.8, 2.6).

filename Each file in UNIX has a name consisting of up to 14 characters and not including the character '/' which is used in *pathname* building. Most *filenames* do not begin with the character '.', and contain only letters and digits with perhaps a '.' separating the *base* portion of the *filename* from an *extension* (1.6).

filename expansion

Filename expansion uses the metacharacters '*', '?', and '[' and ']' to provide a convenient mechanism for naming files. Using *filename expansion* it is easy to name all the files in the current directory, or all files which have a common *root* name. Other *filename expansion* mechanisms use the metacharacter '~' and allow files in other users' directories to be named easily (1.6, 4.2).

flag Many UNIX commands accept arguments which are not the names of files or other users but are used to modify the action of the commands. These are referred to as *flag* options, and by convention consist of one or more letters preceded by the character '-' (1.2). Thus the *ls* (list files) command has an option '-s' to list the sizes of files. This is specified

```
ls -s
```

foreach The *foreach* command is used in shell scripts and at the terminal to specify repetition of a sequence of commands while the value of a certain shell variable ranges through a specified list (3.6, 4.1).

foreground When commands are executing in the normal way such that the shell is waiting for them to finish before prompting for another command they are said to be *foreground jobs* or *running in the foreground*. This is as opposed to *background*. *Foreground* jobs can be stopped by signals from the terminal caused by typing different control characters at the keyboard (1.8, 2.6).

goto The shell has a command *goto* used in shell scripts to transfer control to a given label (3.7).

grep The *grep* command searches through a list of argument files for a specified string. Thus

```
grep bill /etc/passwd
```

will print each line in the file */etc/passwd* which contains the string 'bill'. Actually, *grep* scans for *regular expressions* in the sense of the editors 'ed (1)' and 'ex (1)'. *Grep* stands for 'globally find *regular expression* and print' (2.4).

head The *head* command prints the first few lines of one or more files. If you have a bunch of files containing text which you are wondering about it is sometimes useful to run *head* with these files as arguments. This will usually show enough of what is in these files to let you decide which you are interested in (1.5).

Head is also used to describe the part of a *pathname* before and including the last '/' character. The *tail* of a *pathname* is the part after the last '/'. The 'h' and 't' modifiers allow the *head* or *tail* of a *pathname* stored in a shell variable to be used (3.6).

history The *history* mechanism of the shell allows previous commands to be repeated, possibly after modification to correct typing mistakes or to change the meaning of the command. The shell has a *history list* where these commands are kept, and a *history*

- file after you `logout` (2.1).
- `logout` The `logout` command causes a login shell to exit. Normally, a login shell will exit when you hit control-d generating an *end-of-file*, but if you have set `ignoreeof` in your `.login` file then this will not work and you must use `logout` to log off the UNIX system (2.8).
- `.logout` When you log off of UNIX the shell will execute commands from the file `.logout` in your *home* directory after it prints 'logout'.
- `lpr` The command `lpr` is the line printer daemon. The standard input of `lpr` spooled and printed on the UNIX line printer. You can also give `lpr` a list of filenames as arguments to be printed. It is most common to use `lpr` as the last component of a *pipeline* (2.3).
- `ls` The `ls` (list files) command is one of the most commonly used UNIX commands. With no argument filenames it prints the names of the files in the current directory. It has a number of useful *flag* arguments, and can also be given the names of directories as arguments, in which case it lists the names of the files in these directories (1.2).
- `mail` The `mail` program is used to send and receive messages from other UNIX users (1.1, 2.1), whether they are logged on or not.
- `make` The `make` command is used to maintain one or more related files and to organize functions to be performed on these files. In many ways `make` is easier to use, and more helpful than shell command scripts (3.2).
- `makefile` The file containing commands for `make` is called `makefile` or `Makefile` (3.2).
- `manual` The `manual` often referred to is the 'UNIX manual'. It contains 8 numbered sections with a description of each UNIX program (section 1), system call (section 2), subroutine (section 3), device (section 4), special data structure (section 5), game (section 6), miscellaneous item (section 7) and system administration program (section 8). There are also supplementary documents (tutorials and reference guides) for individual programs which require explanation in more detail. An online version of the `manual` is accessible through the `man` command. Its documentation can be obtained online via
- `man man`
- If you can't decide what manual page to look in, try the `apropos(1)` command. The supplementary documents are in subdirectories of `/usr/doc`.
- `metacharacter` Many characters which are neither letters nor digits have special meaning either to the shell or to UNIX. These characters are called *metacharacters*. If it is necessary to place these characters in arguments to commands without them having their special meaning then they must be *quoted*. An example of a *metacharacter* is the character '>' which is used to indicate placement of output into a file. For the purposes of the *history* mechanism, most unquoted *metacharacters* form separate words (1.4). The appendix to this user's manual lists the *metacharacters* in groups by their function.
- `mkdir` The `mkdir` command is used to create a new directory.
- `modifier` Substitutions with the *history* mechanism, keyed by the character '!' or of variables using the metacharacter '\$', are often subjected to modifications, indicated by placing the character ':' after the substitution and following this with the *modifier* itself. The *command substitution* mechanism can also be used to perform modification in a similar way, but this notation is less clear (3.6).
- `more` The program `more` writes a file on your terminal allowing you to control how much text is displayed at a time. `More` can move through the file screenful by screenful, line by line, search forward for a string, or start again at the beginning of the file. It is generally the easiest way of viewing a file (1.8).

- type its name, forgetting the name of the current *working directory* before doing so (2.7).
- port** The part of a computer system to which each terminal is connected is called a *port*. Usually the system has a fixed number of *ports*, some of which are connected to telephone lines for dial-up access, and some of which are permanently wired directly to specific terminals.
- pr** The *pr* command is used to prepare listings of the contents of files with headers giving the name of the file and the date and time at which the file was last modified (2.3).
- printenv** The *printenv* command is used to print the current setting of variables in the environment (2.8).
- process** An instance of a running program is called a *process* (2.6). UNIX assigns each *process* a unique number when it is started – called the *process number*. *Process numbers* can be used to stop individual *processes* using the *kill* or *stop* commands when the *processes* are part of a detached *background* job.
- program** Usually synonymous with *command*; a binary file or shell command script which performs a useful function is often called a *program*.
- prompt** Many programs will print a *prompt* on the terminal when they expect input. Thus the editor 'ex (1)' will print a ':' when it expects input. The shell *prompts* for input with '%' and occasionally with '?' when reading commands from the terminal (1.1). The shell has a variable *prompt* which may be set to a different value to change the shell's main *prompt*. This is mostly used when debugging the shell (2.8).
- pushd** The *pushd* command, which means 'push directory', changes the shell's *working directory* and also remembers the current *working directory* before the change is made, allowing you to return to the same directory via the *popd* command later without retyping its name (2.7).
- ps** The *ps* command is used to show the processes you are currently running. Each process is shown with its unique process number, an indication of the terminal name it is attached to, an indication of the state of the process (whether it is running, stopped, awaiting some event (sleeping), and whether it is swapped out), and the amount of CPU time it has used so far. The command is identified by printing some of the words used when it was invoked (2.6). Shells, such as the *csi* you use to run the *ps* command, are not normally shown in the output.
- pwd** The *pwd* command prints the full *pathname* of the current *working directory*. The *dirs* builtin command is usually a better and faster choice.
- quit** The *quit* signal, generated by a control-\, is used to terminate programs which are behaving unreasonably. It normally produces a core image file (1.8).
- quotation** The process by which metacharacters are prevented their special meaning, usually by using the character '' in pairs, or by using the character '\', is referred to as *quotation* (1.7).
- redirection** The routing of input or output from or to a file is known as *redirection* of input or output (1.3).
- rehash** The *rehash* command tells the shell to rebuild its internal table of which commands are found in which directories in your *path*. This is necessary when a new program is installed in one of these directories (2.8).
- relative pathname** A *pathname* which does not begin with a '/' is called a *relative pathname* since it is interpreted *relative* to the current *working directory*. The first *component* of such a *pathname* refers to some file or directory in the *working directory*, and subsequent *components* between '/' characters refer to directories below the *working directory*.

stty	The <i>stty</i> program changes certain parameters inside UNIX which determine how your terminal is handled. See 'stty (1)' for a complete description (2.6).
substitution	The shell implements a number of <i>substitutions</i> where sequences indicated by meta-characters are replaced by other sequences. Notable examples of this are history <i>substitution</i> keyed by the metacharacter '!' and variable <i>substitution</i> indicated by '\$'. We also refer to <i>substitutions</i> as <i>expansions</i> (3.4).
suspended	A job becomes <i>suspended</i> after a STOP signal is sent to it, either by typing a <i>control-z</i> at the terminal (for <i>foreground</i> jobs) or by using the <i>stop</i> command (for <i>background</i> jobs). When <i>suspended</i> , a job temporarily stops running until it is restarted by either the <i>fg</i> or <i>bg</i> command (2.6).
switch	The <i>switch</i> command of the shell allows the shell to select one of a number of sequences of commands based on an argument string. It is similar to the <i>switch</i> statement in the language C (3.7).
termination	When a command which is being executed finishes we say it undergoes <i>termination</i> or <i>terminates</i> . Commands normally terminate when they read an <i>end-of-file</i> from their <i>standard input</i> . It is also possible to terminate commands by sending them an <i>interrupt</i> or <i>quit</i> signal (1.8). The <i>kill</i> program terminates specified jobs (2.6).
then	The <i>then</i> command is part of the shell's 'if-then-else-endif' control construct used in command scripts (3.6).
time	The <i>time</i> command can be used to measure the amount of CPU and real time consumed by a specified command as well as the amount of disk I/O (input/output), memory used, and number of page faults and swaps taken by the command (2.1, 2.8).
tset	The <i>tset</i> program is used to set standard erase and kill characters and to tell the system what kind of terminal you are using. It is often invoked in a <i>.login</i> file (2.1).
tty	The word <i>tty</i> is a historical abbreviation for 'teletype' which is frequently used in UNIX to indicate the <i>port</i> to which a given terminal is connected. The <i>tty</i> command will print the name of the <i>tty</i> or <i>port</i> to which your terminal is presently connected.
unalias	The <i>unalias</i> command removes aliases (2.8).
UNIX	UNIX is an operating system on which <i>csh</i> runs. UNIX provides facilities which allow <i>csh</i> to invoke other programs such as editors and text formatters which you may wish to use.
unset	The <i>unset</i> command removes the definitions of shell variables (2.2, 2.8).
variable expansion	See <i>variables</i> and <i>expansion</i> (2.2, 3.4).
variables	<i>Variables</i> in <i>csh</i> hold one or more strings as value. The most common use of <i>variables</i> is in controlling the behavior of the shell. See <i>path</i> , <i>noclobber</i> , and <i>ignoreeof</i> for examples. <i>Variables</i> such as <i>argv</i> are also used in writing shell programs (shell command scripts) (2.2).
verbose	The <i>verbose</i> shell variable can be set to cause commands to be echoed after they are history expanded. This is often useful in debugging shell scripts. The <i>verbose</i> variable is set by the shell's <i>-v</i> command line option (3.10).
wc	The <i>wc</i> program calculates the number of characters, words, and lines in the files whose names are given as arguments (2.6).
while	The <i>while</i> builtin control construct is used in shell command scripts (3.7).
word	A sequence of characters which forms an argument to a command is called a <i>word</i> . Many characters which are neither letters, digits, '-', '.', nor '/' form <i>words</i> all by themselves even if they are not surrounded by blanks. Any sequence of characters may be made into a <i>word</i> by surrounding it with "" characters except for the characters "" and '!' which require special treatment (1.1). This process of placing special

DC - An Interactive Desk Calculator

Robert Morris

Lorinda Cherry

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

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.

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.

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.

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

!

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.

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 to 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 $\text{sqrt}(y)$ is Newton's method with successive approximations by the rule

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

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

Push-Down Registers and Arrays

These commands were designed for use 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**. **Sx** pushes the top value of the main stack onto the stack for the register *x*. **Lx** 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.

BC – An Arbitrary Precision Desk-Calculator Language

Lorinda Cherry

Robert Morris

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

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.

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.

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

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.



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.

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.

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.

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.



Appendix

1. Notation

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

2. Tokens

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

2.1. Comments

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

2.2. 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.

2.3. Keywords

The following are reserved keywords:

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

2.4. 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.

3. 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.

3.2. Unary operators

The unary operators bind right to left.

3.2.1. $-expression$

The result is the negative of the expression.

3.2.2. $++named-expression$

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

3.2.3. $--named-expression$

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

3.2.4. $named-expression++$

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

3.2.5. $named-expression--$

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

3.3. Exponentiation operator

The exponentiation operator binds right to left.

3.3.1. $expression \wedge expression$

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:

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

3.4. Multiplicative operators

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

3.4.1. $expression * expression$

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:

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

3.4.2. $expression / expression$

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

3.4.3. $expression \% expression$

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**

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.

6. Statements

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

6.1. 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.

6.2. Compound statements

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

6.3. Quoted string statements

"any string"

This statement prints the string inside the quotes.

6.4. If statements

if(relation) statement

The substatement is executed if the relation is true.

6.5. While statements

while(relation) statement

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

6.6. 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.

6.7. Break statements

break

break causes termination of a for or while statement.

MAIL REFERENCE MANUAL

Kurt Shoens

Revised by

Craig Leres

Version 5.2

August 26, 1986

1. Introduction

Mail provides a simple and friendly environment for sending and receiving mail. It divides incoming mail into its constituent messages and allows the user to deal with them in any order. In addition, it provides a set of *ed*-like commands for manipulating messages and sending mail. *Mail* offers the user simple editing capabilities to ease the composition of outgoing messages, as well as providing the ability to define and send to names which address groups of users. Finally, *Mail* is able to send and receive messages across such networks as the ARPANET, UUCP, and Berkeley network.

This document describes how to use the *Mail* program to send and receive messages. The reader is not assumed to be familiar with other message handling systems, but should be familiar with the UNIX¹ shell, the text editor, and some of the common UNIX commands. “The UNIX Programmer’s Manual,” “An Introduction to Csh,” and “Text Editing with Ex and Vi” can be consulted for more information on these topics.

Here is how messages are handled: the mail system accepts incoming *messages* for you from other people and collects them in a file, called your *system mailbox*. When you login, the system notifies you if there are any messages waiting in your system mailbox. If you are a *csh* user, you will be notified when new mail arrives if you inform the shell of the location of your mailbox. On version 7 systems, your system mailbox is located in the directory `/usr/spool/mail` in a file with your login name. If your login name is “sam,” then you can make *csh* notify you of new mail by including the following line in your `.cshrc` file:

```
set mail=/usr/spool/mail/sam
```

When you read your mail using *Mail*, it reads your system mailbox and separates that file into the individual messages that have been sent to you. You can then read, reply to, delete, or save these messages. Each message is marked with its author and the date they sent it.

¹ UNIX is a trademark of Bell Laboratories.

From root Wed Sep 21 09:21:45 1978
Subject: Tuition fees
Status: R

Tuition fees are due next Wednesday. Don't forget!!

Many *Mail* commands that operate on messages take a message number as an argument like the **type** command. For these commands, there is a notion of a current message. When you enter the *Mail* program, the current message is initially the first one. Thus, you can often omit the message number and use, for example,

t

to type the current message. As a further shorthand, you can type a message by simply giving its message number. Hence,

1

would type the first message.

Frequently, it is useful to read the messages in your mailbox in order, one after another. You can read the next message in *Mail* by simply typing a newline. As a special case, you can type a newline as your first command to *Mail* to type the first message.

If, after typing a message, you wish to immediately send a reply, you can do so with the **reply** command. **Reply**, like **type**, takes a message number as an argument. *Mail* then begins a message addressed to the user who sent you the message. You may then type in your letter in reply, followed by a <control-d> at the beginning of a line, as before. *Mail* will type EOT, then type the ampersand prompt to indicate its readiness to accept another command. In our example, if, after typing the first message, you wished to reply to it, you might give the command:

reply

Mail responds by typing:

To: root
Subject: Re: Tuition fees

and waiting for you to enter your letter. You are now in the message collection mode described at the beginning of this section and *Mail* will gather up your message up to a control-d. Note that it copies the subject header from the original message. This is useful in that correspondence about a particular matter will tend to retain the same subject heading, making it easy to recognize. If there are other header fields in the message, the information found will also be used. For example, if the letter had a "To:" header listing several recipients, *Mail* would arrange to send your reply to the same people as well. Similarly, if the original message contained a "Cc:" (carbon copies to) field, *Mail* would send your reply to those users, too. *Mail* is careful, though, not too send the message to *you*, even if you appear in the "To:" or "Cc:" field, unless you ask to be included explicitly. See section 4 for more details.

After typing in your letter, the dialog with *Mail* might look like the following:

reply
To: root
Subject: Tuition fees

Thanks for the reminder
EOT
&

Another adaptation to user needs that *Mail* provides is that of *aliases*. An alias is simply a name which stands for one or more real user names. *Mail* sent to an alias is really sent to the list of real users associated with it. For example, an alias can be defined for the members of a project, so that you can send mail to the whole project by sending mail to just a single name. The *alias* command in *Mail* defines an alias. Suppose that the users in a project are named Sam, Sally, Steve, and Susan. To define an alias called "project" for them, you would use the *Mail* command:

```
alias project sam sally steve susan
```

The *alias* command can also be used to provide a convenient name for someone whose user name is inconvenient. For example, if a user named "Bob Anderson" had the login name "anderson," you might want to use:

```
alias bob anderson
```

so that you could send mail to the shorter name, "bob."

While the *alias* and *set* commands allow you to customize *Mail*, they have the drawback that they must be retyped each time you enter *Mail*. To make them more convenient to use, *Mail* always looks for two files when it is invoked. It first reads a system wide file "/usr/lib/Mail.rc," then a user specific file, ".mailrc," which is found in the user's home directory. The system wide file is maintained by the system administrator and contains *set* commands that are applicable to all users of the system. The ".mailrc" file is usually used by each user to set options the way he likes and define individual aliases. For example, my .mailrc file looks like this:

```
set ask nosave SHELL=/bin/csh
```

As you can see, it is possible to set many options in the same *set* command. The "nosave" option is described in section 5.

Mail aliasing is implemented at the system-wide level by the mail delivery system *sendmail*. These aliases are stored in the file /usr/lib/aliases and are accessible to all users of the system. The lines in /usr/lib/aliases are of the form:

```
alias: name1, name2, name3
```

where *alias* is the mailing list name and the *name_i* are the members of the list. Long lists can be continued onto the next line by starting the next line with a space or tab. Remember that you must execute the shell command *newaliases* after editing /usr/lib/aliases since the delivery system uses an indexed file created by *newaliases*.

We have seen that *Mail* can be invoked with command line arguments which are people to send the message to, or with no arguments to read mail. Specifying the *-f* flag on the command line causes *Mail* to read messages from a file other than your system mailbox. For example, if you have a collection of messages in the file "letters" you can use *Mail* to read them with:

```
% Mail -f letters
```

You can use all the *Mail* commands described in this document to examine, modify, or delete messages from your "letters" file, which will be rewritten when you leave *Mail* with the *quit* command described below.

Since mail that you read is saved in the file *mbox* in your home directory by default, you can read *mbox* in your home directory by using simply

```
% Mail -f
```

Normally, messages that you examine using the *type* command are saved in the file "mbox" in your home directory if you leave *Mail* with the *quit* command described below. If you wish to retain a message in your system mailbox you can use the *preserve* command to

`copy +classwork`

copies the current message into the *classwork* folder and leaves a copy in your system mailbox.

The `folder` command can be used to direct *Mail* to the contents of a different folder. For example,

`folder +classwork`

directs *Mail* to read the contents of the *classwork* folder. All of the commands that you can use on your system mailbox are also applicable to folders, including `type`, `delete`, and `reply`. To inquire which folder you are currently editing, use simply:

`folder`

To list your current set of folders, use the `folders` command.

To start *Mail* reading one of your folders, you can use the `-f` option described in section 2. For example:

`% Mail -f +classwork`

will cause *Mail* to read your *classwork* folder without looking at your system mailbox.

escape. *Mail* will print out the number of lines and characters written to the file, after which you may continue appending text to your message. Shell metacharacters may be used in the filename, as in `~r` and are expanded with the conventions of your shell.

If you are sending mail from within *Mail's* command mode you can read a message sent to you into the message you are constructing with the escape:

```
~m 4
```

which will read message 4 into the current message, shifted right by one tab stop. You can name any non-deleted message, or list of messages. Messages can also be forwarded without shifting by a tab stop with `~f`. This is the usual way to forward a message.

If, in the process of composing a message, you decide to add additional people to the list of message recipients, you can do so with the escape

```
~t name1 name2 ...
```

You may name as few or many additional recipients as you wish. Note that the users originally on the recipient list will still receive the message; you cannot remove someone from the recipient list with `~t`.

If you wish, you can associate a subject with your message by using the escape

```
~s Arbitrary string of text
```

which replaces any previous subject with "Arbitrary string of text." The subject, if given, is sent near the top of the message prefixed with "Subject:" You can see what the message will look like by using `~p`.

For political reasons, one occasionally prefers to list certain people as recipients of carbon copies of a message rather than direct recipients. The escape

```
~c name1 name2 ...
```

adds the named people to the "Cc:" list, similar to `~t`. Again, you can execute `~p` to see what the message will look like.

The recipients of the message together constitute the "To:" field, the subject the "Subject:" field, and the carbon copies the "Cc:" field. If you wish to edit these in ways impossible with the `~t`, `~s`, and `~c` escapes, you can use the escape

```
~h
```

which prints "To:" followed by the current list of recipients and leaves the cursor (or print-head) at the end of the line. If you type in ordinary characters, they are appended to the end of the current list of recipients. You can also use your erase character to erase back into the list of recipients, or your kill character to erase them altogether. Thus, for example, if your erase and kill characters are the standard (on printing terminals) `#` and `@` symbols,

```
~h
```

```
To: root kurt####bill
```

would change the initial recipients "root kurt" to "root bill." When you type a newline, *Mail* advances to the "Subject:" field, where the same rules apply. Another newline brings you to the "Cc:" field, which may be edited in the same fashion. Another newline leaves you appending text to the end of your message. You can use `~p` to print the current text of the header fields and the body of the message.

To effect a temporary escape to the shell, the escape

```
~!command
```

is used, which executes *command* and returns you to mailing mode without altering the text of your message. If you wish, instead, to filter the body of your message through a shell command, then you can use

and so on. It is actually a feature of UUCP that the map of all the systems in the network is not known anywhere (except where people decide to write it down for convenience). Talk to your system administrator about good ways to get places; the *uname* command will tell you systems whose names are recognized, but not which ones are frequently called or well-connected.

When you use the **reply** command to respond to a letter, there is a problem of figuring out the names of the users in the "To:" and "Cc:" lists *relative to the current machine*. If the original letter was sent to you by someone on the local machine, then this problem does not exist, but if the message came from a remote machine, the problem must be dealt with. *Mail* uses a heuristic to build the correct name for each user relative to the local machine. So, when you **reply** to remote mail, the names in the "To:" and "Cc:" lists may change somewhat.

4.3. Special recipients

As described previously, you can send mail to either user names or *alias* names. It is also possible to send messages directly to files or to programs, using special conventions. If a recipient name has a '/' in it or begins with a '+', it is assumed to be the path name of a file into which to send the message. If the file already exists, the message is appended to the end of the file. If you want to name a file in your current directory (ie, one for which a '/' would not usually be needed) you can precede the name with './' So, to send mail to the file "memo" in the current directory, you can give the command:

```
% Mail ./memo
```

If the name begins with a '+,' it is expanded into the full path name of the folder name in your folder directory. This ability to send mail to files can be used for a variety of purposes, such as maintaining a journal and keeping a record of mail sent to a certain group of users. The second example can be done automatically by including the full pathname of the record file in the *alias* command for the group. Using our previous *alias* example, you might give the command:

```
alias project sam sally steve susan /usr/project/mail_record
```

Then, all mail sent to "project" would be saved on the file "/usr/project/mail_record" as well as being sent to the members of the project. This file can be examined using *Mail -f*.

It is sometimes useful to send mail directly to a program, for example one might write a project billboard program and want to access it using *Mail*. To send messages to the billboard program, one can send mail to the special name '|billboard' for example. *Mail* treats recipient names that begin with a '|' as a program to send the mail to. An *alias* can be set up to reference a '|' prefaced name if desired. *Caveats*: the shell treats '|' specially, so it must be quoted on the command line. Also, the '| program' must be presented as a single argument to mail. The safest course is to surround the entire name with double quotes. This also applies to usage in the *alias* command. For example, if we wanted to alias 'rmsgs' to 'rmsgs -s' we would need to say:

```
alias rmsgs "| rmsgs -s"
```

Print Like **print**, but also print out ignored header fields. See also **print and ignore**.

Reply

Note the capital R in the name. Frame a reply to a one or more messages. The reply (or replies if you are using this on multiple messages) will be sent **ONLY** to the person who sent you the message (respectively, the set of people who sent the messages you are replying to). You can add people using the `~t` and `~c` tilde escapes. The subject in your reply is formed by prefacing the subject in the original message with "Re:" unless it already began thus. If the original message included a "reply-to" header field, the reply will go *only* to the recipient named by "reply-to." You type in your message using the same conventions available to you through the **mail** command. The **Reply** command is especially useful for replying to messages that were sent to enormous distribution groups when you really just want to send a message to the originator. Use it often.

Type Identical to the **Print** command.

alias Define a name to stand for a set of other names. This is used when you want to send messages to a certain group of people and want to avoid retyping their names. For example

```
alias project john sue willie kathryn
```

creates an alias *project* which expands to the four people John, Sue, Willie, and Kathryn.

alternates

If you have accounts on several machines, you may find it convenient to use the `/usr/lib/aliases` on all the machines except one to direct your mail to a single account. The **alternates** command is used to inform *Mail* that each of these other addresses is really *you*. *Alternates* takes a list of user names and remembers that they are all actually you. When you **reply** to messages that were sent to one of these alternate names, *Mail* will not bother to send a copy of the message to this other address (which would simply be directed back to you by the alias mechanism). If *alternates* is given no argument, it lists the current set of alternate names. **Alternates** is usually used in the `.mailrc` file.

chdir The **chdir** command allows you to change your current directory. **Chdir** takes a single argument, which is taken to be the pathname of the directory to change to. If no argument is given, **chdir** changes to your home directory.

copy The **copy** command does the same thing that **save** does, except that it does not mark the messages it is used on for deletion when you quit.

delete

Deletes a list of messages. Deleted messages can be reclaimed with the **undelete** command.

dp These

commands delete the current message and print the next message. They are useful for quickly reading and disposing of mail.

edit To edit individual messages using the text editor, the **edit** command is provided. The **edit** command takes a list of messages as described under the **type** command and processes each by writing it into the file `Messagex` where *x* is the message number being edited and executing the text editor on it. When you have edited the message to your satisfaction, write the message out and quit, upon which *Mail* will read the message back and remove the file. **Edit** may be abbreviated to **e**.

else Marks the end of the then-part of an if statement and the beginning of the part to take effect if the condition of the if statement is false.

if Commands in your “.mailrc” file can be executed conditionally depending on whether you are sending or receiving mail with the **if** command. For example, you can do:

```
if receive
    commands...
endif
```

An **else** form is also available:

```
if send
    commands...
else
    commands...
endif
```

Note that the only allowed conditions are **receive** and **send**.

ignore

Add the list of header fields named to the *ignore list*. Header fields in the ignore list are not printed on your terminal when you print a message. This allows you to suppress printing of certain machine-generated header fields, such as *Via* which are not usually of interest. The **Type** and **Print** commands can be used to print a message in its entirety, including ignored fields. If **ignore** is executed with no arguments, it lists the current set of ignored fields.

list List the valid *Mail* commands.

mail Send mail to one or more people. If you have the *ask* option set, *Mail* will prompt you for a subject to your message. Then you can type in your message, using tilde escapes as described in section 4 to edit, print, or modify your message. To signal your satisfaction with the message and send it, type control-d at the beginning of a line, or a . alone on a line if you set the option *dot*. To abort the message, type two interrupt characters (RUBOUT by default) in a row or use the ~q escape.

mbox Indicate that a list of messages be sent to *mbox* in your home directory when you quit. This is the default action for messages if you do *not* have the *hold* option set.

next The **next** command goes to the next message and types it. If given a message list, **next** goes to the first such message and types it. Thus,

```
next root
```

goes to the next message sent by “root” and types it. The **next** command can be abbreviated to simply a newline, which means that one can go to and type a message by simply giving its message number or one of the magic characters “^” “.” or “\$”. Thus,

```
prints the current message and
```

```
4
```

```
prints message 4, as described previously.
```

preserve

Same as **hold**. Cause a list of messages to be held in your system mailbox when you quit.

print Takes a message list and types out each message on the terminal.

quit Leave *Mail* and update the file, folder, or system mailbox you were reading. Messages that you have examined are marked as “read” and messages that existed when you started are marked as “old.” If you were editing your system mailbox and if you have set the binary option *hold*, all messages which have not been deleted, saved, or mboxed will be retained in your system mailbox. If you were editing your system mailbox and

unset Reverse the action of setting a binary or valued option.

visual

It is often useful to be able to invoke one of two editors, based on the type of terminal one is using. To invoke a display oriented editor, you can use the **visual** command. The operation of the **visual** command is otherwise identical to that of the **edit** command.

Both the **edit** and **visual** commands assume some default text editors. These default editors can be overridden by the valued options “EDITOR” and “VISUAL” for the standard and screen editors. You might want to do:

```
set EDITOR=/usr/ucb/ex VISUAL=/usr/ucb/vi
```

write The **save** command always writes the entire message, including the headers, into the file. If you want to write just the message itself, you can use the **write** command. The **write** command has the same syntax as the **save** command, and can be abbreviated to simply **w**. Thus, we could write the second message by doing:

```
w 2 file.c
```

As suggested by this example, the **write** command is useful for such tasks as sending and receiving source program text over the message system.

z *Mail* presents message headers in windowfuls as described under the **headers** command. You can move *Mail's* attention forward to the next window by giving the

```
z+
```

command. Analogously, you can move to the previous window with:

```
z-
```

5.3. Custom options

Throughout this manual, we have seen examples of binary and valued options. This section describes each of the options in alphabetical order, including some that you have not seen yet. To avoid confusion, please note that the options are either all lower case letters or all upper case letters. When I start a sentence such as: “Ask” causes *Mail* to prompt you for a subject header, I am only capitalizing “ask” as a courtesy to English.

EDITOR

The valued option “EDITOR” defines the pathname of the text editor to be used in the **edit** command and **~e**. If not defined, a standard editor is used.

SHELL

The valued option “SHELL” gives the path name of your shell. This shell is used for the **!** command and **~!** escape. In addition, this shell expands file names with shell metacharacters like ***** and **?** in them.

VISUAL

The valued option “VISUAL” defines the pathname of your screen editor for use in the **visual** command and **~v** escape. A standard screen editor is used if you do not define one.

append

The “append” option is binary and causes messages saved in *mbox* to be appended to the end rather than prepended. Normally, *Mail* will *mbox* in the same order that the system puts messages in your system mailbox. By setting “append,” you are requesting that *mbox* be appended to regardless. It is in any event quicker to append.

ask “Ask” is a binary option which causes *Mail* to prompt you for the subject of each message you send. If you respond with simply a newline, no subject field will be sent.

noheader

The binary option "noheader" suppresses the printing of the version and headers when *Mail* is first invoked. Setting this option is the same as using `-N` on the command line.

nosave

Normally, when you abort a message with two RUBOUTs, *Mail* copies the partial letter to the file "dead.letter" in your home directory. Setting the binary option "nosave" prevents this.

quiet The binary option "quiet" suppresses the printing of the version when *Mail* is first invoked, as well as printing the for example "Message 4:" from the `type` command.

record

If you love to keep records, then the valued option "record" can be set to the name of a file to save your outgoing mail. Each new message you send is appended to the end of the file.

screen

When *Mail* initially prints the message headers, it determines the number to print by looking at the speed of your terminal. The faster your terminal, the more it prints. The valued option "screen" overrides this calculation and specifies how many message headers you want printed. This number is also used for scrolling with the `z` command.

sendmail

To alternate delivery system, set the "sendmail" option to the full pathname of the program to use. Note: this is not for everyone! Most people should use the default delivery system.

toplines

The valued option "toplines" defines the number of lines that the "top" command will print out instead of the default five lines.

verbose

The binary option "verbose" causes *Mail* to invoke sendmail with the `-v` flag, which causes it to go into verbose mode and announce expansion of aliases, etc. Setting the "verbose" option is equivalent to invoking *Mail* with the `-v` flag as described in section 6.

Following the *from* line are zero or more *header field* lines. Each header field line is of the form:

name: information

Name can be anything, but only certain header fields are recognized as having any meaning. The recognized header fields are: *article-id*, *bcc*, *cc*, *from*, *reply-to*, *sender*, *subject*, and *to*. Other header fields are also significant to other systems; see, for example, the current Arpanet message standard for much more on this topic. A header field can be continued onto following lines by making the first character on the following line a space or tab character.

If any headers are present, they must be followed by a blank line. The part that follows is called the *body* of the message, and must be ASCII text, not containing null characters. Each line in the message body must be terminated with an ASCII newline character and no line may be longer than 512 characters. If binary data must be passed through the mail system, it is suggested that this data be encoded in a system which encodes six bits into a printable character. For example, one could use the upper and lower case letters, the digits, and the characters comma and period to make up the 64 characters. Then, one can send a 16-bit binary number as three characters. These characters should be packed into lines, preferably lines about 70 characters long as long lines are transmitted more efficiently.

The message delivery system always adds a blank line to the end of each message. This blank line must not be deleted.

The UUCP message delivery system sometimes adds a blank line to the end of a message each time it is forwarded through a machine.

It should be noted that some network transport protocols enforce limits to the lengths of messages.

9. Summary of commands, options, and escapes

This section gives a quick summary of the *Mail* commands, binary and valued options, and tilde escapes.

The following table describes the commands:

Command	Description
!	Single command escape to shell
-	Back up to previous message
Print	Type message with ignored fields
Reply	Reply to author of message only
Type	Type message with ignored fields
alias	Define an alias as a set of user names
alternates	List other names you are known by
chdir	Change working directory, home by default
copy	Copy a message to a file or folder
delete	Delete a list of messages
dt	Delete current message, type next message
endif	End of conditional statement; see if
edit	Edit a list of messages
else	Start of else part of conditional; see if
exit	Leave mail without changing anything
file	Interrogate/change current mail file
folder	Same as file
folders	List the folders in your folder directory
from	List headers of a list of messages
headers	List current window of messages
help	Print brief summary of <i>Mail</i> commands
hold	Same as preserve
if	Conditional execution of <i>Mail</i> commands
ignore	Set/examine list of ignored header fields
list	List valid <i>Mail</i> commands
local	List other names for the local host
mail	Send mail to specified names
mbox	Arrange to save a list of messages in <i>mbox</i>
next	Go to next message and type it
preserve	Arrange to leave list of messages in system mailbox
quit	Leave <i>Mail</i> ; update system mailbox, <i>mbox</i> as appropriate
reply	Compose a reply to a message
save	Append messages, headers included, on a file
set	Set binary or valued options
shell	Invoke an interactive shell
top	Print first so many (5 by default) lines of list of messages
type	Print messages
undelete	Undelete list of messages
unset	Undo the operation of a set
visual	Invoke visual editor on a list of messages
write	Append messages to a file, don't include headers
z	Scroll to next/previous screenful of headers

The following table shows the command line flags that *Mail* accepts:

Flag	Description
-N	Suppress the initial printing of headers
-T <i>file</i>	Article-id's of read/deleted messages to <i>file</i>
-d	Turn on debugging
-f <i>file</i>	Show messages in <i>file</i> or <i>~/mbox</i>
-h <i>number</i>	Pass on hop count for mail forwarding
-i	Ignore tty interrupt signals
-n	Inhibit reading of <i>/usr/lib/Mail.rc</i>
-r <i>name</i>	Pass on <i>name</i> for mail forwarding
-s <i>string</i>	Use <i>string</i> as subject in outgoing mail
-u <i>name</i>	Read <i>name's</i> mail instead of your own
-v	Invoke sendmail with the <i>-v</i> flag

Notes: *-T*, *-d*, *-h*, and *-r* are not for human use.

**THE RAND MH
MESSAGE HANDLING
SYSTEM:
USER'S MANUAL**

UCI/UCB Version

Marshall T. Rose
John L. Romine

Based on the original manual by
Borden, Gaines, and Shapiro

April 20, 1986
6.4 #2[UCI]

PICK	47
PREV	51
PROMPTER	52
RCVSTORE	54
REFILE	55
REPL	57
RMF	60
RMM	61
SCAN	62
SEND	64
SHOW	66
SORTM	68
VMH	69
WHATNOW	71
WHOM	73
MORE DETAILS	74
MH-ALIAS	75
MH-FORMAT	78
MH-MAIL	82
MH-PROFILE	85
AP	91
CONFLICT	93
DP	94
INSTALL-MH	95
POST	96
5. REPORTING PROBLEMS	98
6. ADVANCED FEATURES	99
USER-DEFINED SEQUENCES	99
Pick and User-Defined Sequences	99
Mark and User-Defined Sequences	100
Public and Private User-Defined Sequences	100
Sequence Negation	100
The Previous Sequence	101
The Unseen Sequence	101
COMPOSITION OF MAIL	101
The Draft Folder	102
What Happens if the Draft Exists	103
The Push Option at What now? Level	104
Options at What now? Level	104
Digests	104
FOLDER HANDLING	105
Relative Folder Addressing	106

READ THIS

Although the *MH* system was originally developed by the Rand Corporation, and is now in the public domain, the Rand Corporation assumes no responsibility for *MH* or this particular version of *MH*.

In addition, the Regents of the University of California issue the following disclaimer in regard to the UCI/UCB version of *MH*:

“Although each program has been tested by its contributor, no warranty, express or implied, is made by the contributor or the University of California, as to the accuracy and functioning of the program and related program material, nor shall the fact of distribution constitute any such warranty, and no responsibility is assumed by the contributor or the University of California in connection herewith.”

This version of *MH* is in the public domain, and as such, there are no real restrictions on its use. The *MH* source code and documentation have no licensing restrictions whatsoever. As a courtesy, the authors ask only that you provide appropriate credit to the Rand Corporation and the University of California for having developed the software.

MH is a software package that is supported neither by the Rand Corporation nor the University of California. However, since we do use the software ourselves and plan to continue using (and improving) *MH*, bug reports and their associated fixes should be reported back to us so that we may include them in future releases. The current computer mailbox for *MH* is **Bug-MH@UCI.EDU** (in the ARPA Internet), and **...!ucbvax!ucivax!bug-mh** (UUCP). Presently, there are two Internet discussion groups, **MH-Users@UCI.EDU** and **MH-Workers@UCI.EDU**. If there is sufficient interest, corresponding Usenet news groups may be established along with the appropriate gateways.

ACKNOWLEDGMENTS

The *MH* system described herein is based on the original Rand *MH* system. It has been extensively developed (perhaps too much so) by Marshall T. Rose and John L. Romine at the University of California, Irvine. Einar A. Stefferud, Jerry N. Sweet, and Terry P. Domae provided numerous suggestions to improve the UCI version of *MH*. Of course, a large number of people have helped *MH* along. The list of "*MH* immortals" is too long to list here. However, Van Jacobson deserves a special acknowledgement for his tireless work in improving the performance of *MH*. Some programs have been speeded-up by a factor of 10 or 20. All of users of *MH*, everywhere, owe a special thanks to Van.

This manual is based on the original *MH* manual written at Rand by Bruce Borden, Stockton Gaines, and Norman Shapiro.

SUMMARY

Electronic communication of text messages is becoming commonplace. Computer-based message systems—software packages that provide tools for dealing with messages—are used in many contexts. In particular, message systems are becoming increasingly important in command and control and intelligence applications.

This report describes a message handling system called *MH*. This system provides the user with tools to compose, send, receive, store, retrieve, forward, and reply to messages. *MH* has been built on the UNIX time-sharing system, a popular operating system developed for the DEC PDP-11 and VAX classes of computers.

A complete description of *MH* is given for users of the system. For those who do not intend to use the system, this description gives a general idea of what a message system is like. The system involves some new ideas about how large subsystems can be constructed.

The interesting and unusual features of *MH* include the following: The user command interface to *MH* is the UNIX “shell” (the standard UNIX command interpreter). Each separable component of message handling, such as message composition or message display, is a separate command. Each program is driven from and updates a private user environment, which is stored as a file between program invocations. This private environment also contains information to “custom tailor” *MH* to the individual's tastes. *MH* stores each message as a separate file under UNIX, and it utilizes the tree-structured UNIX file system to organize groups of files within separate directories or “folders”. All of the UNIX facilities for dealing with files and directories, such as renaming, copying, deleting, cataloging, off-line printing, etc., are applicable to messages and directories of messages (folders). Thus, important capabilities needed in a message system are available in *MH* without the need (often seen in other message systems) for code that duplicates the facilities of the supporting operating system. It also allows users familiar with the shell to use *MH* with minimal effort.

1. INTRODUCTION

Although people can travel cross-country in hours and can reach others by telephone in seconds, communications still depend heavily upon paper, most of which is distributed through the mails.

There are several major reasons for this continued dependence on written documents. First, a written document may be proofread and corrected prior to its distribution, giving the author complete control over his words. Thus, a written document is better than a telephone conversation in this respect. Second, a carefully written document is far less likely to be misinterpreted or poorly translated than a phone conversation. Third, a signature offers reasonable verification of authorship, which cannot be provided with media such as telegrams.

However, the need for fast, accurate, and reproducible document distribution is obvious. One solution in widespread use is the telefax. Another that is rapidly gaining popularity is electronic mail. Electronic mail is similar to telefax in that the data to be sent are digitized, transmitted via phone lines, and turned back into a document at the receiver. The advantage of electronic mail is in its compression factor. Whereas a telefax must scan a page in very fine lines and send all of the black and white information, electronic mail assigns characters fixed codes which can be transmitted as a few bits of information. Telefax presently has the advantage of being able to transmit an arbitrary page, including pictures, but electronic mail is beginning to deal with this problem. Electronic mail also integrates well with current directions in office automation, allowing documents prepared with sophisticated equipment at one site to be quickly transferred and printed at another site.

Currently, most electronic mail is intraorganizational, with mail transfer remaining within one computer. As computer networking becomes more common, however, it is becoming more feasible to communicate with anyone whose computer can be linked to your own via a network.

The pioneering efforts on general-purpose electronic mail were by organizations using the DoD ARPAnet[1]. The capability to send messages between computers existed before the ARPAnet was developed, but it was used only in limited ways. With the advent of the ARPAnet, tools began to be developed which made it convenient for individuals or organizations to distribute messages over broad geographic areas, using diverse computer facilities. The interest and activity in message systems has now reached such proportions that steps have been taken within the DoD to coordinate and unify the development of military message systems. The use of electronic mail is expected to increase dramatically in the next few years. The utility of such systems in the command and control and intelligence environments is clear, and applications in these areas will probably lead the way. As the costs for sending and handling electronic messages continue their rapid decrease, such uses can be expected to spread rapidly into other areas and, of course, will not be limited to the DoD.

A message system provides tools that help users (individuals or organizations) deal with messages in various ways. Messages must be composed, sent, received, stored, retrieved, forwarded, and replied to. Today's best interactive computer systems provide a variety of word-processing and information handling capabilities. The message handling facilities should be well integrated with the rest of the system, so as to be a graceful extension of overall system capability.

The message system described in this report, *MH*, provides most of the features that can be found in other message systems and also incorporates some new ones. It has been built on the UNIX time-sharing system[2], a popular operating system for the DEC PDP-11¹ and VAX-11 classes of computers. A "secure" operating system similar to UNIX is currently being developed[3], and that system will also run *MH*.

¹ PDP and VAX are trademarks of Digital Equipment Corporation.

2. OVERVIEW

There are three main aspects of *MH* : the way messages are stored (the message database), the user's profile (which directs how certain actions of the message handler take place), and the commands for dealing with messages.

Under *MH*, each message is stored as a separate file. A user can take any action with a message that he could with an ordinary file in UNIX. A UNIX directory in which messages are stored is called a folder. Each folder contains some standard entries to support the message-handling functions. The messages in a folder have numerical names. These folders (directories) are entries in a particular directory path, described in the user profile, through which *MH* can find message folders. Using the UNIX "link" facility, it is possible for one copy of a message to be "filed" in more than one folder, providing a message index facility. Also, using the UNIX tree-structured file system, it is possible to have a folder within a folder, nested arbitrarily deep, and have the full power of the *MH* commands available.

Each user of *MH* has a user profile, a file in his \$HOME (initial login) directory called *.mh_profile*. This profile contains several pieces of information used by the *MH* commands: a path name to the directory that contains the message folders and parameters that tailor *MH* commands to the individual user's requirements. There is also another file, called the user context, which contains information concerning which folder the user last referenced (the "current" folder). It also contains most of the necessary state information concerning how the user is dealing with his messages, enabling *MH* to be implemented as a set of individual UNIX commands, in contrast to the usual approach of a monolithic subsystem.

In *MH*, incoming mail is appended to the end of a file in a system spooling area for the user. This area is called the mail drop directory, and the file is called the user's mail drop. Normally when the user logs in, s/he is informed of new mail (or the *MH* program *msgchk* may be run). The user adds the new messages to his/her collection of *MH* messages by invoking the command *inc*. The *inc* (incorporate) command adds the new messages to a folder called "inbox", assigning them names which are consecutive integers starting with the next highest integer available in inbox. *inc* also produces a *scan* summary of the messages thus incorporated. A folder can be compacted into a single file, for easy storage, by using the *packf* command. Also, messages within a folder can be sorted by date and time with the *sortm* command.

There are four commands for examining the messages in a folder: *show*, *prev*, *next*, and *scan*. The *show* command displays a message in a folder, *prev* displays the message preceding the current message, and *next* displays the message following the current message. *MH* lets the user choose the program that displays individual messages. A special program, *mhl*, can be used to display messages according to the user's preferences. The *scan* command summarizes the messages in a folder, normally producing one line per message, showing who the message is from, the date, the subject, etc.

The user may move a message from one folder to another with the command *refile*. Messages may be removed from a folder by means of the command *rmm*. In addition, a user may query what the current folder is and may specify that a new folder become the current folder, through the command *folder*. All folders may be summarized with the *folders* command. A message folder (or subfolder) may be removed by means of the command *rmf*.

A set of messages based on content may be selected by use of the command *pick*. This command searches through messages in a folder and selects those that match a given set of criteria. These messages are then bound to a "sequence" name for use with other *MH* commands. The *mark* command manipulates these sequences.

3. TUTORIAL

This tutorial provides a brief introduction to the *MH* commands. It should be sufficient to allow the user to read his mail, do some simple manipulations of it, and create and send messages.

A message has two major pieces: the header and the body. The body consists of the text of the message (whatever you care to type in). It follows the header and is separated from it by an empty line. (When you compose a message, the form that appears on your terminal shows a line of dashes after the header. This is for convenience and is replaced by an empty line when the message is sent.) The header is composed of several components, including the subject of the message and the person to whom it is addressed. Each component starts with a name and a colon; components must not start with a blank. The text of the component may take more than one line, but each continuation line must start with a blank. Messages typically have "To:", "cc:", and "Subject:" components. When composing a message, you should include the "To:" and "Subject:" components; the "cc:" (for people you want to send copies to) is not necessary.

The basic *MH* commands are *inc*, *scan*, *show*, *next*, *prev*, *rmm*, *comp*, and *repl*. These are described below.

inc

When you get the message "You have mail", type the command *inc*. You will get a "scan listing" such as:

```
7+ 7/13 Cas      revival of measurement work
8  10/ 9 Norm    NBS people and publications
9  11/26 To:norm question <<Are there any functions
```

This shows the messages you received since the last time you executed this command (*inc* adds these new messages to your inbox folder). You can see this list again, plus a list of any other messages you have, by using the *scan* command.

scan

The scan listing shows the message number, followed by the date and the sender. (If you are the sender, the addressee in the "To:" component is displayed. You may send yourself a message by including your name among the "To:" or "cc:" addressees.) It also shows the message's subject; if the subject is short, the first part of the body of the message is included after the characters <<.

show

This command shows the current message, that is, the first one of the new messages after an *inc*. If the message is not specified by name (number), it is generally the last message referred to by an *MH* command. For example,

```
show 5      will show message 5.
```

You can use the show command to copy a message or print a message.

4. DETAILED DESCRIPTION

This section describes the *MH* system in detail, including the components of the user profile, the conventions for message naming, and some of the other *MH* conventions. Readers who are generally familiar with computer systems will be able to follow the principal ideas, although some details may be meaningful only to those familiar with UNIX.

THE USER PROFILE

The first time an *MH* command is issued by a new user, the system prompts for a “Path” and creates an *MH* “profile”.

Each *MH* user has a profile which contains tailoring information for each individual program. Other profile entries control the *MH* path (where folders and special files are kept), folder and message protections, editor selection, and default arguments for each *MH* program. Each user of *MH* also has a context file which contains current state information for the *MH* package (the location of the context file is kept in the user's *MH* directory, or may be named in the user profile). When a folder becomes the current folder, it is recorded in the user's context. (Other state information is kept in the context file, see the manual entry for *mh-profile* (5) for more details.) In general, the term “profile entry” refer to entries in either the profile or context file. Users of *MH* needn't worry about the distinction, *MH* handles these things automatically.

The *MH* profile is stored in the file *.mh_profile* in the user's **\$HOME** directory¹. It has the format of a message without any body. That is, each profile entry is on one line, with a keyword followed by a colon (:) followed by text particular to the keyword.

■ *This file must not have blank lines.*

The keywords may have any combination of upper and lower case. (See the information of *mh-mail* later on in this manual for a description of message formats.)

For the average *MH* user, the only profile entry of importance is “Path”. Path specifies a directory in which *MH* folders and certain files such as “draft” are found. The argument to this keyword must be a legal UNIX path that names an existing directory. If this path is not absolute (i.e., does not begin with a /), it will be presumed to start from the user's **\$HOME** directory. All folder and message references within *MH* will relate to this path unless full path names are used.

Message protection defaults to 644, and folder protection to 711. These may be changed by profile entries “Msg-Protect” and “Folder-Protect”, respectively. The argument to these keywords is an octal number which is used as the UNIX file mode².

When an *MH* program starts running, it looks through the user's profile for an entry with a keyword matching the program's name. For example, when *comp* is run, it looks for a “comp” profile entry. If one is found, the text of the profile entry is used as the default switch setting until all defaults are overridden by explicit switches passed to the program as arguments. Thus the profile entry “comp: -form standard.list” would direct *comp* to use the file “standard.list” as the message skeleton. If an explicit form switch is given to the *comp* command, it will override the switch obtained from the profile.

¹ By defining the environment variable **\$MH**, you can specify an alternate profile to be used by *MH* commands.

² See *chmod* (1) in the *UNIX Programmer's Manual* [5].

Table 1
PROFILE COMPONENTS

Keyword and Argument	MH Programs that use Component
Path: Mail	All
Current-Folder: inbox	Most
Editor: /usr/ucb/ex	<i>comp, dist, forw, repl</i>
Msg-Protect: 644	<i>inc</i>
Folder-Protect: 711	<i>inc, pick, refile</i>
<program>: default switches	All
prompter-next: ed	<i>comp, dist, forw, repl</i>

Path should be present. Current-Folder is maintained automatically by many *MH* commands (see the **Context** sections of the individual commands in Sec. IV). All other entries are optional, defaulting to the values described above.

MESSAGE NAMING

Messages may be referred to explicitly or implicitly when using *MH* commands. A formal syntax of message names is given in Appendix B, but the following description should be sufficient for most *MH* users. Some details of message naming that apply only to certain commands are included in the description of those commands.

Most of the *MH* commands accept arguments specifying one or more folders, and one or more messages to operate on. The use of the word "msg" as an argument to a command means that exactly one message name may be specified. A message name may be a number, such as 1, 33, or 234, or it may be one of the "reserved" message names: first, last, prev, next, and cur. (As a shorthand, a period (.) is equivalent to cur.) The meanings of these names are straightforward: "first" is the first message in the folder; "last" is the last message in the folder; "prev" is the message numerically previous to the current message; "next" is the message numerically following the current message; "cur" (or ".") is the current message in the folder. In addition, *MH* supports user-defined-sequences; see the description of the *mark* command for more information.

The default in commands that take a "msg" argument is always "cur".

The word "msgs" indicates that several messages may be specified. Such a specification consists of several message designations separated by spaces. A message designation is either a message name or a message range. A message range is a specification of the form name1-name2 or name1:n, where name1 and name2 are message names and n is an integer. The first form designates all the messages from name1 to name2 inclusive; this must be a non-empty range. The second form specifies up to n messages, starting with name1 if name1 is a number, or first, cur, or next, and ending with name1 if name1 is last or prev. This interpretation of n is overridden if n is preceded by a plus sign or a minus sign; +n always means up to n messages starting with name1, and -n always means up to n messages ending with name1. Repeated specifications of the same message have the same effect as a single specification of the message. Examples of specifications are:

MH COMMANDS

The *MH* package comprises several programs:

ali (1)	- list mail aliases
anno (1)	- annotate messages
burst (1)	- explode digests into messages
comp (1)	- compose a message
dist (1)	- redistribute a message to additional addresses
folder (1)	- set/list current folder/message
folders (1)	- list all folders
forw (1)	- forward messages
inc (1)	- incorporate new mail
mark (1)	- mark messages
mhl (1)	- produce formatted listings of MH messages
mhmail (1)	- send or read mail
mhook (1)	- MH receive-mail hooks
mhpath (1)	- print full pathnames of MH messages and folders
msgchk (1)	- check for messages
msh (1)	- MH shell (and BBoard reader)
next (1)	- show the next message
packf (1)	- compress a folder into a single file
pick (1)	- select messages by content
prev (1)	- show the previous message
prompter (1)	- prompting editor front end
rcvstore (1)	- incorporate new mail asynchronously
refile (1)	- file messages in other folders
repl (1)	- reply to a message
rmf (1)	- remove folder
rmm (1)	- remove messages
scan (1)	- produce a one line per message scan listing
send (1)	- send a message
show (1)	- show (list) messages
sortm (1)	- sort messages
vmh (1)	- visual front-end to MH
whatnow (1)	- prompting front-end for send
whom (1)	- report to whom a message would go

These programs are described below. The form of the descriptions conforms to the standard form for the description of UNIX commands.

NAME

anno – annotate messages

SYNOPSIS

anno [+folder] [msgs] [--component field] [--inplace] [--noinplace] [--text body] [--help]

DESCRIPTION

Anno annotates the specified messages in the named folder using the field and body. Annotation is optionally performed by *dist*, *forw*, and *repl*, to keep track of your distribution of, forwarding of, and replies to a message. By using *anno*, you can perform arbitrary annotations of your own. Each message selected will be annotated with the lines

```
field: date
field: body
```

The ‘--inplace’ switch causes annotation to be done in place in order to preserve links to the annotated message.

The field specified should be a valid 822-style message field name, which means that it should consist of alphanumerics (or dashes) only. The body specified is arbitrary text.

If a ‘--component field’ is not specified when *anno* is invoked, *anno* will prompt the user for the name of field for the annotation.

Files

\$HOME/.mh_profile The user profile

Profile Components

Path: To determine the user’s MH directory
Current-Folder: To find the default current folder

See Also

dist (1), forw (1), repl (1)

Defaults

‘+folder’ defaults to the current folder
‘msgs’ defaults to cur
‘--noinplace’

Context

If a folder is given, it will become the current folder. The first message annotated will become the current message.

Context

If a folder is given, it will become the current folder. If '-inplace' is given, then the first message burst becomes the current message. This leaves the context ready for a *show* of the table of contents of the digest, and a *next* to see the first message of the digest. If '-noinplace' is given, then the first message extracted from the first digest burst becomes the current message. This leaves the context in a similar, but not identical, state to the context achieved when using '-inplace'.

Bugs

The *burst* program enforces a limit on the number of messages which may be *burst* from a single message. This number is on the order of 1000 messages. There is usually no limit on the number of messages which may reside in the folder after the *bursting*.

Although *burst* uses a sophisticated algorithm to determine where one encapsulated message ends and another begins, not all digestifying programs use an encapsulation algorithm. In degenerate cases, this usually results in *burst* finding an encapsulation boundary prematurely and splitting a single encapsulated message into two or more messages. These erroneous digestifying programs should be fixed.

Furthermore, any text which appears after the last encapsulated message is not placed in a separate message by *burst*. In the case of digestified messages, this text is usually an "End of digest" string. As a result of this possibly un-friendly behavior on the part of *burst*, note that when the '-inplace' option is used, this trailing information is lost. In practice, this is not a problem since correspondents usually place remarks in text prior to the first encapsulated message, and this information is not lost.

Profile Components

Path:	To determine the user's MH directory
Draft-Folder:	To find the default draft-folder
Editor:	To override the default editor
Msg-Protect:	To set mode when creating a new message (draft)
fileproc:	Program to refile the message
whatnowproc:	Program to ask the "What now?" questions

See Also

dist(1), forw(1), repl(1), send(1), whatnow(1)

Defaults

'+folder' defaults to the current folder
'msg' defaults to the current message
'-nodraftfolder'
'-nouse'

Context

None

Bugs

If *whatnowproc* is *whatnow*, then *comp* uses a built-in *whatnow*, it does not actually run the *whatnow* program. Hence, if you define your own *whatnowproc*, don't call it *whatnow* since *comp* won't run it.

Upon exiting from the editor, *dist* will invoke the *whatnow* program. See *whatnow* (1) for a discussion of available options. The invocation of this program can be inhibited by using the ‘-nowhatnowproc’ switch. (In truth of fact, it is the *whatnow* program which starts the initial edit. Hence, ‘-nowhatnowproc’ will prevent any edit from occurring.)

Files

/usr/new/lib/mh/distcomps	The message skeleton
or <mh-dir>/distcomps	Rather than the standard skeleton
\$HOME/.mh_profile	The user profile
<mh-dir>/draft	The draft file

Profile Components

Path:	To determine the user’s MH directory
Current-Folder:	To find the default current folder
Draft-Folder:	To find the default draft-folder
Editor:	To override the default editor
fileproc:	Program to refile the message
whatnowproc:	Program to ask the “What now?” questions

See Also

comp(1), forw(1), repl(1), send(1), whatnow(1)

Defaults

‘+folder’ defaults to the current folder
 ‘msg’ defaults to cur
 ‘-noannotate’
 ‘-nodraftfolder’
 ‘-noinplace’

Context

If a folder is given, it will become the current folder. The message distributed will become the current message.

History

Dist originally used headers of the form “Distribute-xxx:” instead of “Resent-xxx:”. In order to conform with the ARPA Internet standard, RFC-822, the “Resent-xxx:” form is now used. *Dist* will recognize “Distribute-xxx:” type headers and automatically convert them to “Resent-xxx:”.

Bugs

Dist does not *rigorously* check the message being distributed for adherence to the transport standard, but *post* called by *send* does. The *post* program will balk (and rightly so) at poorly formatted messages, and *dist* won’t correct things for you.

If *whatnowproc* is *whatnow*, then *dist* uses a built-in *whatnow*, it does not actually run the *whatnow* program. Hence, if you define your own *whatnowproc*, don’t call it *whatnow* since *dist* won’t run it.

If your current working directory is not writable, the link named “@” is not available.

If '-fast' is given, only the folder name (or names in the case of '-all') will be listed. (This is faster because the folders need not be read.)

The '-pack' switch will compress the message names in a folder, removing holes in message numbering.

The '-recurse' switch will list each folder recursively. Use of this option effectively defeats the speed enhancement of the '-fast' option, since each folder must be searched for subfolders. Nevertheless, the combination of these options is useful.

If the specified (or default) folder doesn't exist, the user will be queried if the folder should be created. (This is the easy way to create an empty folder for use later.)

The '-push' switch directs *folder* to push the current folder onto the *folder-stack*, and make the '+folder' argument the current folder. If '+folder' is not given, the current folder and the top of the *folder-stack* are exchanged. This corresponds to the "pushd" operation in the *CShell*.

The '-pop' switch directs *folder* to discard the top of the *folder-stack*, after setting the current folder to that value. No '+folder' argument is allowed. This corresponds to the "popd" operation in the *CShell*. The '-push' switch and the '-pop' switch are mutually exclusive: the last occurrence of either one overrides any previous occurrence of the other.

The '-list' switch directs *folder* to list the contents of the *folder-stack*. No '+folder' argument is allowed. After a successful '-push' or '-pop', the '-list' action is taken. This corresponds to the "dirs" operation in the *CShell*.

Files

\$HOME/.mh_profile The user profile

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder
Folder-Protect:	To set mode when creating a new folder
Folder-Stack:	To determine the folder stack
lsproc:	Program to list the contents of a folder

See Also

refile(1), mhpath(1)

Defaults

'+folder' defaults to the current folder
 'msg' defaults to none
 '-nofast'
 '-noheader'
 '-nototal'
 '-nopack'
 '-norecurse'
 '-print' is the default if no '-list', '-push', or '-pop' is specified

Context:

If '+folder' and/or 'msg' are given, they will become the current folder and/or message.

NAME

forw – forward messages

SYNOPSIS

forw [+folder] [msgs] [-annotate] [-noannotate] [-draftfolder +folder] [-draftmessage msg]
 [-nodraftfolder] [-editor editor] [-noedit] [-filter filterfile] [-form formfile] [-format]
 [-noformat] [-inplace] [-noinplace] [-whatnowproc program] [-nowhatnowproc]
 [-help]

forw [+folder] [msgs] [-digest list] [-issue number] [-volume number]
 [other switches for *forw*] [-help]

DESCRIPTION

Forw may be used to prepare a message containing other messages. It constructs the new message from the components file or '-form formfile' (see *comp*), with a body composed of the message(s) to be forwarded. An editor is invoked as in *comp*, and after editing is complete, the user is prompted before the message is sent.

The default message form contains the following elements:

```
To:
cc:
Subject:
-----
```

If the file named "forwcomps" exists in the user's MH directory, it will be used instead of this form. In either case, the file specified by '-form formfile' will be used if given.

If the draft already exists, *forw* will ask you as to the disposition of the draft. A reply of **quit** will abort *forw*, leaving the draft intact; **replace** will replace the existing draft with a blank skeleton; and **list** will display the draft.

If the '-annotate' switch is given, each message being forwarded will be annotated with the lines

```
Forwarded: date
Forwarded: addr
```

where each address list contains as many lines as required. This annotation will be done only if the message is sent directly from *forw*. If the message is not sent immediately from *forw*, "comp -use" may be used to re-edit and send the constructed message, but the annotations won't take place. The '-inplace' switch causes annotation to be done in place in order to preserve links to the annotated message.

See *comp* (1) for a description of the '-editor' and '-noedit' switches.

Although *forw* uses the '-form formfile' switch to direct it how to construct the beginning of the draft, the '-filter filterfile', '-format', and '-noformat' switches direct *forw* as to how each forwarded message should be formatted in the body of the draft. If '-noformat' is specified, then each forwarded message is output exactly as it appears. If '-format' or '-filter filterfile' is specified, then each forwarded message is filtered (re-formatted) prior to being output to the body of the draft. The filter file for *forw* should be a standard form file for *mhl*, as *forw* will

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder
Draft-Folder:	To find the default draft-folder
Editor:	To override the default editor
Msg-Protect:	To set mode when creating a new message (draft)
fileproc:	Program to refile the message
mhproc:	Program to filter messages being forwarded
whatnowproc:	Program to ask the "What now?" questions

See Also

Proposed Standard for Message Encapsulation (aka RFC-934),
comp(1), dist(1), repl(1), send(1), whatnow(1)

Defaults

'+folder' defaults to the current folder
'msgs' defaults to cur
'-noannotate'
'-nodraftfolder'
'-noformat'
'-noinplace'

Context

If a folder is given, it will become the current folder. The first message forwarded will become the current message.

Bugs

If *whatnowproc* is *whatnow*, then *forw* uses a built-in *whatnow*, it does not actually run the *whatnow* program. Hence, if you define your own *whatnowproc*, don't call it *whatnow* since *forw* won't run it.

When *forw* is told to annotate the messages it forwards, it doesn't actually annotate them until the draft is successfully sent. If from the *whatnowproc*, you *push* instead of *send*, it's possible to confuse *forw* by re-ordering the file (e.g., by using 'folder -pack') before the message is successfully sent. *Dist* and *repl* don't have this problem.

If the environment variable `$MAILDROP` is set, then *inc* uses it as the location of the user's maildrop instead of the default (the `-file name` switch still overrides this, however). If this environment variable is not set, then *inc* will consult the profile entry "MailDrop" for this information. If the value found is not absolute, then it is interpreted relative to the user's *MH* directory. If the value is not found, then *inc* will look in the standard system location for the user's maildrop.

The `-silent` switch directs *inc* to be quiet and not ask any questions at all. This is useful for putting *inc* in the background and going on to other things.

Files

<code>\$HOME/mh_profile</code>	The user profile
<code>/usr/new/lib/mh/mtstailor</code>	tailor file
<code>/usr/spool/mail/\$USER</code>	Location of mail drop

Profile Components

Path:	To determine the user's MH directory
Alternate-Mailboxes:	To determine the user's mailboxes
Folder-Protect:	To set mode when creating a new folder
Msg-Protect:	To set mode when creating a new message and audit-file
Unseen-Sequence:	To name sequences denoting unseen messages

See Also

`mhmail(1)`, `scan(1)`, `mh-mail(5)`, `post(8)`

Defaults

- `+folder` defaults to "inbox"
- `-noaudit`
- `-changezur`
- `-format` defaulted as described above
- `-nosilent`
- `-truncate` if `-file name` not given, `-nottruncate` otherwise
- `-width` defaulted to the width of the terminal

Context

The folder into which messages are being incorporated will become the current folder. The first message incorporated will become the current message, unless the `-nochangezur` option is specified. This leaves the context ready for a *show* of the first new message.

Bugs

The argument to the `-format` switch must be interpreted as a single token by the shell that invokes *inc*. Therefore, one must usually place the argument to this switch inside double-quotes.

usually limited to 10.

The name used to denote a message sequence can not occur as part of a message range, e.g., constructs like "seen:20" or "seen-10" are forbidden.

Files

\$HOME/.mh_profile The user profile

Profile Components

Path: To determine the user's MH directory
Current-Folder: To find the default current folder

See Also

pick (1)

Defaults

'+folder' defaults to the current folder
'-add' if 'msgs' is specified, '-list' otherwise
'msgs' defaults to cur (or all if '-list' is specified)
'-npublic' if the folder is read-only, '-public' otherwise
'-nozero'

Context

If a folder is given, it will become the current folder.

the order in the format file.

Each line of mhl.format has one of the formats:

```
;comment
:cleartext
variable[,variable...]
component:[variable,...]
```

A line beginning with a ';' is a comment, and is ignored. A line beginning with a ':' is clear text, and is output exactly as is. A line containing only a ':' produces a blank line in the output. A line beginning with "component:" defines the format for the specified component, and finally, remaining lines define the global environment.

For example, the line:

```
width=80,length=40,clearscreen,overflowtext="****",overflowoffset=5
```

defines the screen size to be 80 columns by 40 rows, specifies that the screen should be cleared prior to each page, that the overflow indentation is 5, and that overflow text should be flagged with "****".

Following are all of the current variables and their arguments. If they follow a component, they apply only to that component, otherwise, their affect is global. Since the whole format is parsed before any output processing, the last global switch setting for a variable applies to the whole message if that variable is used in a global context (i.e., bell, clearscreen, width, length).

<i>variable</i>	<i>type</i>	<i>semantics</i>
width	integer	screen width or component width
length	integer	screen length or component length
offset	integer	positions to indent "component:"
overflowtext	string	text to use at the beginning of an overflow line
overflowoffset	integer	positions to indent overflow lines
compwidth	integer	positions to indent component text after the first line is output
uppercase	flag	output text of this component in all upper case
nouppercase	flag	don't uppercase
clearscreen	flag/G	clear the screen prior to each page
noclearscreen	flag/G	don't clearscreen
bell	flag/G	ring the bell at the end of each page
nobell	flag/G	don't bell
component	string/L	name to use instead of "component" for this component
nocomponent	flag	don't output "component:" for this component
center	flag	center component on line (works for one-line components only)
nocenter	flag	don't center
leftadjust	flag	strip off leading whitespace on each line of text
noleftadjust	flag	don't leftadjust

mhl can be given a default format string for either address or date fields (but not both). To do this, on a global line specify either the variable *addrfield* or the variable *datefield*, along with the variable *formatfield*.

Files

<code>/usr/new/lib/mh/mhl.format</code>	The message template
or <code><mh-dir>/mhl.format</code>	Rather than the standard template
<code>\$HOME/.mh_profile</code>	The user profile

Profile Components

<code>moreproc:</code>	Program to use as interactive front-end
------------------------	---

See Also

`show(1)`, `ap(8)`, `dp(8)`

Defaults

`'-bell'`
`'-noclear'`
`'-length 40'`
`'-width 80'`

Context

None

Bugs

There should be some way to pass 'bell' and 'clear' information to the front-end.

On hosts where *MH* was configured with the BERK option, address parsing is not enabled.

NAME

mhook – MH receive-mail hooks

SYNOPSIS

`$HOME/.maildelivery`

`/usr/new/lib/mh/rcvdist address ... [-help]`

`/usr/new/lib/mh/rcvpack file [-help]`

`/usr/new/lib/mh/rcvty [command ...] [-help]`

DESCRIPTION

A receive-mail hook is a program that is run whenever you receive a mail message. You do **NOT** invoke the hook yourself, rather the hook is invoked on your behalf by *SendMail*, when you include the line

“| /usr/new/lib/mh/slocal”

in your `.forward` file in your home directory.

The `.maildelivery` file, which is an ordinary ASCII file, controls how local delivery is performed. This file is read by *slocal*.

The format of each line in the `.maildelivery` file is

field pattern action result string

where

field:

The name of a field that is to be searched for a pattern. This is any field in the headers of the message that might be present. In addition, the following special fields are also defined:

source: the out-of-band sender information

addr: the address that was used to cause delivery to the recipient

default: this matches *only* if the message hasn't been delivered yet

***: this always matches

pattern:

The sequence of characters to match in the specified field. Matching is case-insensitive but not RE-based.

action:

The action to take to deliver the message. This is one of

file or *>*:

Append the message to the file named by *string*. The standard maildrop delivery process is used. If the message can be appended to the file, then this action succeeds.

When writing to the file, a new field is added:

Delivery-Date: date

Perform the action only if the message has not been delivered. If the action succeeded, then the message is considered delivered.

The file is always read completely, so that several matches can be made and several actions can be taken. The *.maildelivery* file must be owned either by the user or by root, and must be writable only by the owner. If the *.maildelivery* file can not be found, or does not perform an action which delivers the message, then the file */usr/new/lib/mh/maildelivery* is read according to the same rules. This file must be owned by the root and must be writable only by the root. If this file can not be found or does not perform an action which delivers the message, then standard delivery to the user's maildrop, */usr/spool/mail/\$USER*, is performed.

Arguments in the *.maildelivery* file are separated by white-space or comma. Since double-quotes are honored, these characters may be included in a single argument by enclosing the entire argument in double-quotes. A double-quote can be included by preceding it with a backslash.

To summarize, here's an example:

```
#field  pattern          action result string
# lines starting with a '#' are ignored, as are blank lines
#
# file mail with mmdf2 in the "To:" line into file mmdf2.log
To      mmdf2            file  A    mmdf2.log
# Messages from mmdf pipe to the program err-message-archive
From    mmdf             pipe  A    err-message-archive
# Anything with the "Sender:" address "uk-mmdf-workers"
# file in mmdf2.log if not filed already
Sender  uk-mmdf-workers  file  ?    mmdf2.log
# "To:" unix - put in file unix-news
To      Unix             >    A    unix-news
# if the address is jpo=mmdf - pipe into mmdf-redist
addr    jpo=mmdf         |    A    mmdf-redist
# if the address is jpo=ack - send an acknowledgement copy back
addr    jpo=ack          |    R    "resend -r $(reply-to)"
# anything from steve - destroy!
From    steve            destroy A  -
# anything not matched yet - put into mailbox
default -                >    ?    mailbox
# always run rcvalert
*      -                 |    R    rcvalert
```

Four programs are currently standardly available, *rcvdist* (redistribute incoming messages to additional recipients), *rcvpack* (save incoming messages in a *packf'd* file), and *rcvitty* (notify user of incoming messages). The fourth program, *rcvstore* (1) is described separately. They all reside in the */usr/new/lib/mh/* directory.

The *rcvdist* program will resend a copy of the message to all of the addresses listed on its command line.

The *rcvpack* program will append a copy of the message to the file listed on its command line. Its use is obsolete by the *.maildelivery*.

NAME

`mhpath` – print full pathnames of MH messages and folders

SYNOPSIS

`mhpath` [+folder] [msgs] [-help]

DESCRIPTION

Mhpath expands and sorts the message list ‘msgs’ and writes the full pathnames of the messages to the standard output separated by newlines. If no ‘msgs’ are specified, *mhpath* outputs the folder pathname instead.

Contrasted with other MH commands, a message argument to *mhpath* may often be intended for *writing*. Because of this: 1) the name “new” has been added to *mhpath*’s list of reserved message names (the others are “first”, “last”, “prev”, “next”, “cur”, and “all”). The new message is equivalent to the message after the last message in a folder (and equivalent to 1 in a folder without messages). The “new” message may not be used as part of a message range. 2) Within a message list, the following designations may refer to messages that do not exist: a single numeric message name, the single message name “cur”, and (obviously) the single message name “new”. All other message designations must refer to at least one existing message. 3) An empty folder is not in itself an error.

Message numbers greater than the highest existing message in a folder as part of a range designation are replaced with the next free message number.

Examples: The current folder foo contains messages 3 5 6. Cur is 4.

```
% mhpath
/r/phyl/Mail/foo
```

```
% mhpath all
/r/phyl/Mail/foo/3
/r/phyl/Mail/foo/5
/r/phyl/Mail/foo/6
```

```
% mhpath 2001
/r/phyl/Mail/foo/7
```

```
% mhpath 1–2001
/r/phyl/Mail/foo/3
/r/phyl/Mail/foo/5
/r/phyl/Mail/foo/6
```

```
% mhpath new
/r/phyl/Mail/foo/7
```

```
% mhpath last new
/r/phyl/Mail/foo/6
/r/phyl/Mail/foo/7
```

```
% mhpath last–new
bad message list “last–new”.
```

NAME

msgchk – check for messages

SYNOPSIS

msgchk [users ...] [-help]

DESCRIPTION

The *msgchk* program checks all known mail drops for mail waiting for you to receive. For those drops which have mail for you, *msgchk* will indicate if it believes that you have seen the mail in question before.

Files

\$HOME/.mh_profile	The user profile
/usr/new/lib/mh/mtstailor	tailor file
/usr/spool/mail/\$USER	Location of mail drop

Profile Components

None

See Also

inc(1)

Defaults

'user' defaults to the current user

Context

None

to *bbc*, and *bbc* will continue examining the list of BBoards that it is scanning.

If the file is writable and has been modified, then using “quit” will query the user if the file should be updated.

The ‘-prompt string’ switch sets the prompting string for *msh*.

You may wish to use an alternate *MH* profile for the commands that *msh* executes; see *mh-profile* (5) for details about the *\$MH* environment variable.

When invoked from *bbc*, two special features are enabled: First, the ‘-scan’ switch directs *msh* to do a ‘scan unseen’ on start-up if new items are present in the BBoard. This feature is best used from *bbc*, which correctly sets the stage. Second, the *mark* command in *msh* acts specially when you are reading a BBoard, since *msh* will consult the sequence “unseen” in determining what messages you have actually read. When *msh* exits, it reports this information to *bbc*. In addition, if you give the *mark* command with no arguments, *msh* will interpret it as ‘mark -sequence unseen -delete -nozero all’ Hence, to discard all of the messages in the current BBoard you’re reading, just use the *mark* command with no arguments.

When invoked from *vmh*, another special feature is enabled: The ‘topcur’ switch directs *msh* to have the current message “track” the top line of the *vmh* scan window. Normally, *msh* has the current message “track” the center of the window (under ‘-notopcur’, which is the default).

msh supports an output redirection facility. Commands may be followed by one of

```
> file      write output to file
>> file    append output to file
| command  pipe output to UNIX command
```

If *file* starts with a ‘ ’ (tilde), then a *cs*h-like expansion takes place. Note that *command* is interpreted by *sh* (1). Also note that *msh* does NOT support history substitutions, variable substitutions, or alias substitutions.

When parsing commands to the left of any redirection symbol, *msh* will honor ‘\’ (back-slash) as the quote next-character symbol, and “” (double-quote) as quote-word delimiters. All other input tokens are separated by whitespace (spaces and tabs).

Files

<i>\$HOME/.mh_profile</i>	The user profile
<i>/usr/new/lib/mh/mtstailor</i>	tailor file

Profile Components

Path:	To determine the user’s MH directory
Msg-Protect:	To set mode when creating a new ‘file’
fileproc:	Program to file messages
showproc:	Program to show messages

See Also

bbc(1)

NAME

`next` – show the next message

SYNOPSIS

`next` [+folder] [-header] [-noheader] [-showproc program] [-noshowproc]
[switches for *showproc*] [-help]

DESCRIPTION

Next performs a *show* on the next message in the specified (or current) folder. Like *show*, it passes any switches on to the program *showproc*, which is called to list the message. This command is almost exactly equivalent to “show next”. Consult the manual entry for *show* (1) for all the details.

Files

`$HOME/.mh_profile` The user profile

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder
showproc:	Program to show the message

See Also

`show(1)`, `prev(1)`

Defaults

'+folder' defaults to the current folder
'-format'
'-header'

Context

If a folder is specified, it will become the current folder. The message that is shown (i.e., the next message in sequence) will become the current message.

Bugs

next is really a link to the *show* program. As a result, if you make a link to *next* and that link is not called *next*, your link will act like *show* instead. To circumvent this, add a profile-entry for the link to your *MH* profile and add the argument *next* to the entry.

NAME

pick – select messages by content

SYNOPSIS

```
pick { -cc           [+folder] [msgs] [-help]
      -date        [-before date] [-after date] [-datefield field]
      -from
      -search      pattern [-and ...] [-or ...] [-not ...] [-lbrace ... -rbrace]
      -subject
      -to          [-sequence name ...] [-public] [-npublic] [-zero] [-nozero]
      --component [-list] [-nolist]
```

typically:

```
scan 'pick -from jones'
pick -to holloway -sequence select
show 'pick -before friday'
```

DESCRIPTION

Pick searches messages within a folder for the specified contents, and then identifies those messages. Two types of search primitives are available: pattern matching and date constraint operations.

A modified *grep*(1) is used to perform the matching, so the full regular expression (see *ed*(1)) facility is available within 'pattern'. With '-search', 'pattern' is used directly, and with the others, the grep pattern constructed is:

```
"component[ \t]*:.*pattern"
```

This means that the pattern specified for a '-search' will be found everywhere in the message, including the header and the body, while the other pattern matching requests are limited to the single specified component. The expression

```
'--component pattern'
```

is a shorthand for specifying

```
'-search "component[ \t]*:.*pattern"'
```

It is used to pick a component which is not one of "To:", "cc:", "Date:", "From:", or "Subject:". An example is 'pick --reply-to pooh'.

Pattern matching is performed on a per-line basis. Within the header of the message, each component is treated as one long line, but in the body, each line is separate. Lower-case letters in the search pattern will match either lower or upper case in the message, while upper case will match only upper case.

Independent of any pattern matching operations requested, the switches '-after date' or '-before date' may also be used to introduce date/time constraints on all of the messages. By default, the "Date:" field is consulted, but if another date yielding field (such as "BB-Posted:" or "Delivery-Date:") should be used, the '-datefield field' switch may be used. *Pick* will actually parse the date fields in each of the messages specified in 'msgs' (unlike the '-date' switch described above which does a pattern matching operation), and compare them to the

Files

\$HOME/.mh_profile .The user profile

Profile Components

Path: To determine the user's MH directory
 Current-Folder: To find the default current folder

See Also

mark(1)

Defaults

'+folder' defaults to the current folder
 'msgs' defaults to all
 '-datefield date'
 '-npublic' if the folder is read-only, '-public' otherwise
 '-zero'
 '-list' is the default if no '-sequence', '-nolist' otherwise

Context

If a folder is given, it will become the current folder.

History

In previous versions of *MH*, the *pick* command would *show*, *scan*, or *refile* the selected messages. This was rather "inverted logic" from the UNIX point of view, so *pick* was changed to define sequences and output those sequences. Hence, *pick* can be used to generate the arguments for all other *MH* commands, instead of giving *pick* endless switches for invoking those commands itself.

Also, previous versions of *pick* balked if you didn't specify a search string or a date/time constraint. The current version does not, and merely matches the messages you specify. This lets you type something like:

```
show 'pick last:20 -seq fear'
```

instead of typing

```
mark -add -nozero -seq fear last:20
show fear
```

Finally, timezones used to be ignored when comparing dates: they aren't any more.

NAME

prev – show the previous message

SYNOPSIS

prev [+folder] [-header] [-noheader] [-showproc program] [-noshowproc]
 [-switches for *showproc*] [-help]

DESCRIPTION

Prev performs a *show* on the previous message in the specified (or current) folder. Like *show*, it passes any switches on to the program named by *showproc*, which is called to list the message. This command is almost exactly equivalent to “show prev”. Consult the manual entry for *show* (1) for all the details.

Files

\$HOME/.mh_profile The user profile

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder
showproc:	Program to show the message

See Also

show(1), next(1)

Defaults

'+folder' defaults to the current folder
 '-format'
 '-header'

Context

If a folder is specified, it will become the current folder. The message that is shown (i.e., the previous message in sequence) will become the current message.

Bugs

prev is really a link to the *show* program. As a result, if you make a link to *prev* and that link is not called *prev*, your link will act like *show* instead. To circumvent this, add a profile-entry for the link to your *MH* profile and add the argument *prev* to the entry.

The first non-flag argument to *prompter* is taken as the name of the draft file, and subsequent non-flag arguments are ignored.

Files

`$HOME/.mh_profile` The user profile
`/tmp/prompter*` Temporary copy of message

Profile Components

`prompter-next:` To name the editor to be used on exit from *prompter*
`Msg-Protect:` To set mode when creating a new draft

See Also

`comp(1)`, `dist(1)`, `forw(1)`, `repl(1)`, `whatnow(1)`

Defaults

`'-prepend'`
`'-norapid'`

Context

None

Bugs

Prompter uses *stdio* (3), so it will lose if you edit files with nulls in them.

NAME

refile - file message in other folders

SYNOPSIS

refile [msgs] [--draft] [--link] [--nolink] [--preserve] [--nopreserve] [--src +folder] [--file file]
+folder ... [--help]

DESCRIPTION

Refile moves (mv (1)) or links (ln (1)) messages from a source folder into one or more destination folders. If you think of a message as a sheet of paper, this operation is not unlike filing the sheet of paper (or copies) in file cabinet folders. When a message is filed, it is linked into the destination folder(s) if possible, and is copied otherwise. As long as the destination folders are all on the same file system, multiple filing causes little storage overhead. This facility provides a good way to cross-file or multiply-index messages. For example, if a message is received from Jones about the ARPA Map Project, the command

```
refile `cur` +jones +Map
```

would allow the message to be found in either of the two folders 'jones' or 'Map'.

The option '-file file' directs refile to use the specified file as the source message to be filed, rather than a message from a folder. Note that the file should be a validly formatted message, just like any other MH message. It should NOT be in mail drop format (to convert a file in mail drop format to a folder of MH messages, see inc (1)).

If a destination folder doesn't exist, refile will ask if you want to create it. A negative response will abort the file operation.

The option '-link' preserves the source folder copy of the message (i.e., it does a ln(1) rather than a mv(1)), whereas, '-nolink' deletes the filed messages from the source folder. Normally, when a message is filed, it is assigned the next highest number available in each of the destination folders. Use of the '-preserve' switch will override this message renaming, but name conflicts may occur, so use this switch cautiously.

If '-link' is not specified (or '-nolink' is specified), the filed messages will be removed (unlink (2)) from the source folder, similar to the way mv (1) works.

If the user has a profile component such as

```
rmmproc: /bin/rm
```

then instead of simply renaming the message file, refile will call the named program to delete the file.

The '-draft' switch tells refile to file the <mh-dir>/draft.

Files

\$HOME/.mh_profile The user profile



NAME

repl – reply to a message

SYNOPSIS

```
repl [+folder] [msg] [-annotate] [-noannotate] [-cc all/to/cc/me] [-nocc all/to/cc/me]
    [-draftfolder +folder] [-draftmessage msg] [-nodraftfolder] [-editor editor] [-noedit]
    [-fcc +folder] [-filter filterfile] [-form formfile] [-format] [-noformat] [-inplace]
    [-notinplace] [-query] [-noquery] [-width columns] [-whatnowproc program]
    [-nowhatnowproc] [-help]
```

DESCRIPTION

Repl aids a user in producing a reply to an existing message. *Repl* uses a reply template to guide its actions when constructing the message draft of the reply. In its simplest form (with no arguments), it will set up a message-form skeleton in reply to the current message in the current folder, and invoke the *whatnow* shell. The default reply template will direct *repl* to construct the composed message as follows:

```
To: <Reply-To> or <From>
cc: <cc>, <To>, and yourself
Subject: Re: <Subject>
In-reply-to: Your message of <Date>.
             <Message-Id>
```

where field names enclosed in angle brackets (< >) indicate the contents of the named field from the message to which the reply is being made. The ‘-cc type’ switch takes an argument which specifies who gets placed on the “cc:” list of the reply. The ‘-query’ switch modifies the action of ‘-cc type’ switch by interactively asking you if each address that normally would be placed in the “To:” and “cc:” list should actually be sent a copy. (This is useful for special-purpose replies.) Note that the position of the ‘-cc’ and ‘-nocc’ switches, like all other switches which take a positive and negative form, is important.

If the file named “replcomps” exists in the user’s MH directory, it will be used instead of the default form. In either case, the file specified by ‘-form formfile’ will be used if given.

If the draft already exists, *repl* will ask you as to the disposition of the draft. A reply of **quit** will abort *repl*, leaving the draft intact; **replace** will replace the existing draft with a blank skeleton; and **list** will display the draft.

See *comp* (1) for a description of the ‘-editor’ and ‘-noedit’ switches. Note that while in the editor, the message being replied to is available through a link named “@” (assuming the default *whatnowproc*). In addition, the actual pathname of the message is stored in the environment variable `$editalt`, and the pathname of the folder containing the message is stored in the environment variable `$mhfolder`.

Although *repl* uses the ‘-form formfile’ switch to direct it how to construct the beginning of the draft, the ‘-filter filterfile’ switch directs *repl* as to how the message being replied-to should be formatted in the body of the draft. If ‘-filter’ is not specified, then the message being replied-to is not included in the body of the draft. If ‘-filter filterfile’ is specified, then the message being replied-to is filtered (re-formatted) prior to being output to the body of the draft. The filter file for *repl* should be a standard form file for *mhl*, as *repl* will invoke *mhl* to format the message being replied-to. There is no default message filter (‘-filter’ must be followed by a file name). A filter file that is commonly used is:

Profile Components

Path:	To determine the user's MH directory
Alternate-Mailboxes:	To determine the user's mailboxes
Current-Folder:	To find the default current folder
Draft-Folder:	To find the default draft-folder
Editor:	To override the default editor
Msg-Protect:	To set mode when creating a new message (draft)
fileproc:	Program to refile the message
mhlproc:	Program to filter message being replied-to
whatnowproc:	Program to ask the "What now?" questions

See Also

comp(1), dist(1), forw(1), send(1), whatnow(1), mh-format(5)

Defaults

'+folder' defaults to the current folder
 'msg' defaults to cur
 '-nocc all' at ATHENA sites, '-cc all' otherwise
 '-format'
 '-noannotate'
 '-nodraftfolder'
 '-noinplace'
 '-noquery'
 '-width 72'

Context

If a folder is given, it will become the current folder. The message replied-to will become the current message.

Bugs

If any addresses occur in the reply template, addresses in the template that do not contain hosts are defaulted incorrectly. Instead of using the localhost for the default, *repl* uses the sender's host. Moral of the story: if you're going to include addresses in a reply template, include the host portion of the address.

If *whatnowproc* is *whatnow*, then *repl* uses a built-in *whatnow*, it does not actually run the *whatnow* program. Hence, if you define your own *whatnowproc*, don't call it *whatnow* since *repl* won't run it.

If your current working directory is not writable, the link named "@" is not available.

NAME

rmm – remove messages

SYNOPSIS

rmm [+folder] [msgs] [-help]

DESCRIPTION

Rmm removes the specified messages by renaming the message files with preceding commas. Many sites consider files that start with a comma to be a temporary backup, and arrange for *cron* (8) to remove such files once a day.

If the user has a profile component such as

```
rmmproc:    /bin/rm
```

then instead of simply renaming the message file, *rmm* will call the named program to delete the file. Note that at most installations, *cron* (8) is told to remove files that begin with a comma once a night.

Some users of *csch* prefer the following:

```
alias rmm 'refile +d'
```

where folder +d is a folder for deleted messages, and

```
alias mexp 'rm `mhpath +d all`'
```

is used to “expunge” deleted messages.

The current message is not changed by *rmm*, so a *next* will advance to the next message in the folder as expected.

Files

\$HOME/.mh_profile The user profile

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder
rmmproc:	Program to delete the message

See Also

rmf(1)

Defaults

'+folder' defaults to the current folder
'msgs' defaults to cur

Context

If a folder is given, it will become the current folder.

simply a format string and the file is simply a format file. See *mh-format* (5) for the details.

In addition to the standard escapes, *scan* also recognizes the following additional escape:

escape substitution

body the (compressed) first part of the body

On hosts where *MH* was configured with the BERK option, *scan* has two other switches: '-reverse', and '-noreverse'. These make *scan* list the messages in reverse order. In addition, *scan* will update the *MH* context prior to starting the listing, so interrupting a long *scan* listing preserves the new context. *MH* purists hate both of these ideas.

Files

\$HOME/.mh_profile The user profile

Profile Components

Path: To determine the user's MH directory
 Alternate-Mailboxes: To determine the user's mailboxes
 Current-Folder: To find the default current folder

See Also

inc(1), pick(1), show(1), mh-format(5)

Defaults

'+folder' defaults to the folder current
 'msgs' defaults to all
 '-format' defaulted as described above
 '-noheader'
 '-width' defaulted to the width of the terminal

Context

If a folder is given, it will become the current folder.

History

Prior to using the format string mechanism, '-header' used to generate a heading saying what each column in the listing was. Format strings prevent this from happening.

Bugs

The argument to the '-format' switch must be interpreted as a single token by the shell that invokes *scan*. Therefore, one must usually place the argument to this switch inside double-quotes.

already contains a "From:" field, then a "Sender: user@local" field will be added as well. (An already existing "Sender:" field is an error!)

By using the '-format' switch, each of the entries in the "To:" and "cc:" fields will be replaced with "standard" format entries. This standard format is designed to be usable by all of the message handlers on the various systems around the Internet. If '-noformat' is given, then headers are output exactly as they appear in the message draft.

If an "Fcc: folder" is encountered, the message will be copied to the specified folder for the sender in the format in which it will appear to any non-Bcc receivers of the message. That is, it will have the appended fields and field reformatting. The "Fcc:" fields will be removed from all outgoing copies of the message.

By using the '-width columns' switch, the user can direct *send* as to how long it should make header lines containing addresses.

By using the '-alias aliasfile' switch, the user can direct *send* to consult the named files for alias definitions (more than one file, each preceded by '-alias', can be named). See *mh-alias* (5) for more information.

Files

\$HOME/.mh_profile The user profile

Profile Components

Path:	To determine the user's MH directory
Draft-Folder:	To find the default draft-folder
Signature:	To determine the user's mail signature
mailproc:	Program to post failure notices
postproc:	Program to post the message

See Also

comp(1), dist(1), forw(1), repl(1), mh-alias(5), post(8)

Defaults

'file' defaults to <mh-dir>/draft
 '-alias /usr/new/lib/mh/MailAliases'
 '-nodraftfolder'
 '-nofilter'
 '-format'
 '-forward'
 '-nomsgid'
 '-nopush'
 '-noverbose'
 '-nowatch'
 '-width 72'

Context

None

Defaults

'+folder' defaults to the current folder
 'msgs' defaults to cur
 '-format'
 '-header'

Context

If a folder is given, it will become the current folder. The last message shown will become the current message.

Bugs

The '-header' switch doesn't work when 'msgs' expands to more than one message. If the *showproc* is *mhl*, then this problem can be circumvented by referencing the "messagename" field in the *mhl* format file.

Show updates the user's context before showing the message. Hence if *show* will mark messages as seen prior to the user actually seeing them. This is generally not a problem, unless the user relies on the "unseen" messages mechanism, and interrupts *show* while it is showing "unseen" messages.

If *showproc* is *mhl*, then *show* uses a built-in *mhl*: it does not actually run the *mhl* program. Hence, if you define your own *showproc*, don't call it *mhl* since *show* won't run it.

If *more* (1) is your *showproc* (the default), then avoid running *show* in the background with only its standard output piped to another process, as in

```
show | imprint &
```

Due to a bug in *more*, *show* will go into a "tty input" state. To avoid this problem, re-direct *show*'s diagnostic output as well. For users of *csh*:

```
show |& imprint &
```

For users of *sh*:

```
show 2>&1 | imprint &
```

NAME

vmh – visual front–end to MH

SYNOPSIS

vmh [-prompt string] [-vmhproc program] [-novmhproc] [switches for *vmhproc*] [-help]

DESCRIPTION

vmh is a program which implements the server side of the *MH* window management protocol and uses *curses* (3) routines to maintain a split–screen interface to any program which implements the client side of the protocol. This latter program, called the *vmhproc*, is specified using the ‘-vmhproc program’ switch.

The upshot of all this is that one can run *msh* on a display terminal and get a nice visual interface. To do this, for example, just add the line

```
mshproc: vmh
```

to your *.mh_profile*. (This takes advantage of the fact that *msh* is the default *vmhproc* for *vmh*.)

In order to facilitate things, if the ‘-novmhproc’ switch is given, and *vmh* can’t run on the user’s terminal, the *vmhproc* is run directly without the window management protocol.

After initializing the protocol, *vmh* prompts the user for a command to be given to the client. Usually, this results in output being sent to one or more windows. If a output to a window would cause it to scroll, *vmh* prompts the user for instructions, roughly permitting the capabilities of *less* or *more* (e.g., the ability to scroll backwards and forwards):

```
SPACE      advance to the next windowful
RETURN *   advance to the next line
y          *   retreat to the previous line
d          *   advance to the next ten lines
u          *   retreat to the previous ten lines
g          *   go to an arbitrary line
            (preceed g with the line number)
G          *   go to the end of the window
            (if a line number is given, this acts like ‘g’)
CTRL-L     refresh the entire screen
h          print a help message
q          abort the window
```

(A ‘*’ indicates that a numeric prefix is meaningful for this command.)

Note that if a command resulted in more than one window’s worth of information being displayed, and you allow the command which is generating information for the window to gracefully finish (i.e., you don’t use the ‘q’ command to abort information being sent to the window), then *vmh* will give you one last change to peruse the window. This is useful for scrolling back and forth. Just type ‘q’ when you’re done.

To abnormally terminate *vmh* (without core dump), use <QUIT> (usually CTRL-^). For instance, this does the “right” thing with *bbc* and *msh*.

NAME

whatnow – prompting front–end for send

SYNOPSIS

whatnow [-draftfolder +folder] [-draftmessage msg] [-nodraftfolder] [-editor editor] [-noedit] [-prompt string] [file] [-help]

DESCRIPTION

Whatnow is the default program that queries the user about the disposition of a composed draft. It is normally invoked by one of *comp*, *dist*, *forw*, or *repl* after the initial edit.

When started, the editor is started on the draft (unless ‘-noedit’ is given, in which case the initial edit is suppressed). Then, *whatnow* repetitively prompts the user with “What now?” and awaits a response. The valid responses are

display to list the message being distributed/replied–to on the terminal
edit to re–edit using the same editor that was used on the preceding round unless a profile entry “<lasteditor>–next: <editor>” names an alternate editor
edit <editor> to invoke <editor> for further editing
list to list the draft on the terminal
push to send the message in the background
quit to terminate the session and preserve the draft
quit –delete to terminate, then delete the draft
refile +folder to refile the draft into the given folder
send to send the message
send –watch to cause the delivery process to be monitored
whom to list the addresses that the message will go to
whom –check to list the addresses and verify that they are acceptable to the transport service

For the **edit** response, any valid switch to the editor is valid. Similarly, for the **send** and **whom** responses, any valid switch to *send* (1) and *whom* (1) commands, respectively, are valid. For the **push** response, any valid switch to *send* (1) is valid (as this merely invokes *send* with the ‘–push’ option). For the **refile** response, any valid switch to the *fileproc* is valid. For the **display** and **list** responses, any valid argument to the *lproc* is valid. If any non–switch arguments are present, then the pathname of the draft will be excluded from the argument list given to the *lproc* (this is useful for listing another *MH* message).

See *mh–profile* (5) for further information about how editors are used by *MH*. It also discusses how complex environment variables can be used to direct *whatnow*’s actions.

The ‘–prompt string’ switch sets the prompting string for *whatnow*.

The ‘–draftfolder +folder’ and ‘–draftmessage msg’ switches invoke the *MH* draft folder facility. This is an advanced (and highly useful) feature. Consult the **Advanced Features** section of the *MH* manual for more information.

Files

\$HOME/.mh_profile	The user profile
<mh-dir>/draft	The draft file

NAME

whom – report to whom a message would go

SYNOPSIS

whom [-alias aliasfile] [-check] [-nocheck] [-draft] [-draftfolder +folder] [-draftmessage msg]
[-nodraftfolder] [file] [-help]

DESCRIPTION

Whom is used to expand the headers of a message into a set of addresses and optionally verify that those addresses are deliverable at that time (if ‘-check’ is given).

The ‘-draftfolder +folder’ and ‘-draftmessage msg’ switches invoke the *MH* draft folder facility. This is an advanced (and highly useful) feature. Consult the **Advanced Features** section of the *MH* manual for more information.

By using the ‘-alias aliasfile’ switch, the user can direct *send* to consult the named files for alias definitions (more than one file, each preceded by ‘-alias’, can be named). See *mh-alias* (5) for more information.

Files

\$HOME/.mh_profile The user profile

Profile Components

Draft-Folder: To find the default draft-folder
postproc: Program to post the message

See Also

mh-alias(5), post(8)

Defaults

‘file’ defaults to <mh-dir>/draft
‘-nocheck’
‘-alias /usr/new/lib/mh/MailAliases’

Context

None

Bugs

With the ‘-check’ option, *whom* makes no guarantees that the addresses listed as being ok are really deliverable, rather, an address being listed as ok means that at the time that *whom* was run the address was thought to be deliverable by the transport service. For local addresses, this is absolute; for network addresses, it means that the host is known; for uucp addresses, it (often) means that the *UUCP* network is available for use.

NAME

mh-alias – alias file for MH message system

SYNOPSIS

any *MH* command

DESCRIPTION

This describes both *MH* personal alias files and the (primary) alias file for mail delivery, the file

`/usr/new/lib/mh/MailAliases`

It does **not** describe alias files used by the message transport system. Each line of the alias file has the format:

```
alias : address-group
or
alias ; address-group
or
< alias-file
```

where:

```
address-group := address-list
                | "<" file
                | "=" UNIX-group
                | "+" UNIX-group
                | "*"
address-list  := address
                | address-list, address
```

Continuation lines in alias files end with ‘\’ followed by the newline character.

Alias-file and file are UNIX file names. UNIX-group is a group name (or number) from */etc/group*. An address is a “simple” Internet-style address. Throughout this file, case is ignored, except for alias-file names.

If the line starts with a ‘<’, then the file named after the ‘<’ is read for more alias definitions. The reading is done recursively, so a ‘<’ may occur in the beginning of an alias file with the expected results.

If the address-group starts with a ‘<’, then the file named after the ‘<’ is read and its contents are added to the address-list for the alias.

If the address-group starts with an ‘=’, then the file */etc/group* is consulted for the UNIX-group named after the ‘=’. Each login name occurring as a member of the group is added to the address-list for the alias.

In contrast, if the address-group starts with a ‘+’, then the file */etc/group* is consulted to determine the group-id of the UNIX-group named after the ‘+’. Each login name occurring in the */etc/passwd* file whose group-id is indicated by this group is added to the address-list

to be local, a system-wide alias file is consulted. These aliases are **NOT** expanded into the headers of messages delivered.

Helpful Hints

To use aliasing in *MH* quickly, do the following:

First, in your *.mh_profile*, choose a name for your primary alias file, say “aliases”, and add three lines:

```
ali: -alias aliases
send: -alias aliases
whom: -alias ailases
```

Second, create the file “aliases” in your *MH* directory.

Third, start adding aliases to your “aliases” file as appropriate.

Files

/usr/new/lib/mh/MailAliases Primary alias file

Profile Components

None

See Also

ali(1), send(1), whom(1), group(5), passwd(5), conflict(8), post(8)

Defaults

None

Context

None

History

In previous releases of *MH*, only a single, system-wide *mh*-alias file was supported. This led to a number of problems, since only mail-system administrators were capable of (un)defining aliases. Hence, the semantics of *mh*-alias were extended to support personal alias files. Users of *MH* no longer need to bother mail-system administrators for keeping information in the system-wide alias file, as each *MH* user can create/modify/remove aliases at will from any number of personal files.

Bugs

Although the forward-referencing semantics of *mh*-alias files prevent recursion, the “< alias-file” command may defeat this. Since the number of file descriptors is finite (and very limited), such infinite recursion will terminate with a meaningless diagnostic when all the fds are used up.

cur		integer	message is current
size		integer	size of message
strlen	string	integer	length of <i>str</i>
me		string	the user's mailbox
plus		integer	add width to <i>num</i>
minus		integer	subtract <i>num</i> from width
charleft		integer	space left in output buffer
timenow		integer	seconds since the UNIX epoch

When *str* is a date, these escapes are useful:

<i>escape</i>	<i>argument</i>	<i>returns</i>	<i>interpretation</i>
sec	string	integer	seconds of the minute
min	string	integer	minutes of the day
hour	string	integer	hours of the day (24 hour clock)
mday	string	integer	day of the month
mon	string	integer	month of the year
wday	string	integer	day of the week (Sunday=0)
year	string	integer	year of the century
yday	string	integer	day of the year
dst	string	integer	daylight savings in effect
zone	string	integer	timezone
sday	string	integer	day of the week known 1 for explicit in date 0 for implicit (<i>MH</i> figured it out) -1 for unknown (<i>MH</i> couldn't figure it out)
clock	string	integer	seconds since the UNIX epoch
rclock	string	integer	seconds prior to current time
month	string	string	month of the year
lmonth	string	string	month of the year (long form)
tzone	string	string	timezone
day	string	string	day of the week
weekday	string	string	day of the week (long)
tws	string	string	official 822 rendering of the date
pretty	string	string	a more user-friendly rendering
nodate	string		date wasn't parseable

When *str* is an address, these escapes are useful:

<i>escape</i>	<i>argument</i>	<i>returns</i>	<i>interpretation</i>
pers	string	string	the personal name of the address
mbox	string	string	the local part of the address
host	string	string	the domain part of the address
path	string	string	the route part of the address
type	string	integer	the type of host -1 for uucp 0 for local 1 for network 2 for unknown
nohost	string	integer	no host was present in the address
ingrp	string	integer	the address appeared inside a group
gname	string	string	name of the group (present for first address only)
note	string	string	commentary text
proper	string	string	official 822 rendering of the address

Context

None

Bugs

On hosts where *MII* was configured with the BERK option, address parsing is not enabled.

programs.

Date:

Added by *post* (8), contains date and time of the message's entry into the transport system.

From:

Added by *post* (8), contains the address of the author or authors (may be more than one if a "Sender:" field is present). Replies are typically directed to addresses in the "Reply-To:" or "From:" field (the former has precedence if present).

Sender:

Added by *post* (8) in the event that the message already has a "From:" line. This line contains the address of the actual sender. Replies are never sent to addresses in the "Sender:" field.

To:

Contains addresses of primary recipients.

cc:

Contains addresses of secondary recipients.

Bcc:

Still more recipients. However, the "Bcc:" line is not copied onto the message as delivered, so these recipients are not listed. *MH* uses an encapsulation method for blind copies, see *send* (1).

Fcc:

Causes *post* (8) to copy the message into the specified folder for the sender, if the message was successfully given to the transport system.

Message-ID:

A unique message identifier added by *post* (8) if the '-msgid' flag is set.

Subject:

Sender's commentary. It is displayed by *scan* (1).

In-Reply-To:

A commentary line added by *repl* (1) when replying to a message.

Resent-Date:

Added when redistributing a message by *post* (8).

Resent-From:

Added when redistributing a message by *post* (8).

Resent-To:

New recipients for a message resent by *dist* (1).

Resent-cc:

Still more recipients. See "cc:" and "Resent-To:".

Resent-Bcc:

Even more recipients. See "Bcc:" and "Resent-To:".

NAME

`.mh_profile` – user customization for MH message system

SYNOPSIS

any *MH* command

DESCRIPTION

Each user of *MH* is expected to have a file named `.mh_profile` in his or her home directory. This file contains a set of user parameters used by some or all of the *MH* family of programs. Each line of the file is of the format

profile-component: value

The possible profile components are exemplified below. Only 'Path:' is mandatory. The others are optional; some have default values if they are not present. In the notation used below, (profile, default) indicates whether the information is kept in the user's *MH* profile or *MH* context, and indicates what the default value is.

Path: Mail

Locates *MH* transactions in directory "Mail". (profile, no default)

context: context

Declares the location of the *MH* context file, see the **HISTORY** section below. (profile, default: <mh-dir>/context)

Current-Folder: inbox

Keeps track of the current open folder. (context, default: +inbox)

Previous-Sequence: pseq

Names the sequences which should be defined as the 'msgs' or 'msg' argument given to the program. If not present, or empty, no sequences are defined. Otherwise, for each name given, the sequence is first zero'd and then each message is added to the sequence. (profile, no default)

Sequence-Negation: not

Defines the string which, when prefixed to a sequence name, negates that sequence. Hence, "notseen" means all those messages that are not a member of the sequence "seen". (profile, no default)

Unseen-Sequence: unseen

Names the sequences which should be defined as those messages recently incorporated by *inc*. *Show* knows to remove messages from this sequence once it thinks they have been seen. If not present, or empty, no sequences are defined. Otherwise, for each name given, the sequence is first zero'd and then each message is added to the sequence. (profile, no default)

mh-sequences: `.mh_sequences`

The name of the file in each folder which defines public sequences. To disable the use of public sequences, leave the value portion of this entry blank. (profile, default: `.mh_sequences`)

atr-seq-folder: 172 178-181 212

default)

digest-issue-list: 1

Tells *forw* the last issue of the last volume sent for the digest *list*. (context, no default)

digest-volume-list: 1

Tells *forw* the last volume sent for the digest *list*. (context, no default)

MailDrop: .mail

Tells *inc* your maildrop, if different from the default. This is superceded by the **\$MAILDROP** environment variable. (profile, default: /usr/spool/mail/\$USER)

Signature: Rand MH System (agent: Marshall Rose)

Tells *send* your mail signature. This is superceded by the **\$SIGNATURE** environment variable. On hosts where *MH* was configured with the UCI option, if **\$SIGNATURE** is not set and this profile entry is not present, the file \$HOME/.signature is consulted. (profile, no default)

The following profile elements are used whenever an *MH* program invokes some other program such as *more* (1). The *.mh_profile* can be used to select alternate programs if the user wishes. The default values are given in the examples.

```
fileproc:      /usr/new/mh/refile
incproc:       /usr/new/mh/inc
installproc:   /usr/new/lib/mh/install-mh
lproc:         /usr/ucb/more
mailproc:      /usr/new/mh/mhmail
mhlproc:       /usr/new/lib/mh/mhl
moreproc:      /usr/ucb/more
mshproc:       /usr/new/mh/msh
packproc:      /usr/new/mh/packf
postproc:      /usr/new/lib/mh/post
rmmproc:       none
rmfproc:       /usr/new/mh/rmf
sendproc:      /usr/new/mh/send
showproc:      /usr/ucb/more
whatnowproc:   /usr/new/mh/whatnow
whomproc:      /usr/new/mh/whom
```

If you define the environment variable **\$MH**, you can specify a profile other than *.mh_profile* to be read by the *MH* programs that you invoke. If the value of **\$MH** is not absolute, (i.e., does not begin with a /), it will be presumed to start from the current working directory. This is one of the very few exceptions in *MH* where non-absolute pathnames are not considered relative to the user's *MH* directory.

Similarly, if you define the environment variable **\$MHCONTEXT**, you can specify a context other than the normal context file (as specified in the *MH* profile). As always, unless the value of **\$MHCONTEXT** is absolute, it will be presumed to start from your *MH* directory.

MH programs also support other environment variables:

If the OVERHEAD option was set during *MH* configuration (type ‘-help’ to an *MH* command to find out), then if this environment variable is set, *MH* considers it to be the number of a file-descriptor which is opened, read-only to the *MH* profile. Similarly, if the environment variable \$MHCONTEXTFD is set, this is the number of a file-descriptor which is opened read-only to the *MH* context. This feature of *MH* is experimental, and is used to examine possible speed improvements for *MH* startup. Note that these environment variables must be set and non-empty to enable this feature. However, if OVERHEAD is enabled during *MH* configuration, then when *MH* programs call other *MH* programs, this scheme is used. These file-descriptors are not closed throughout the execution of the *MH* program, so children may take advantage of this. This approach is thought to be completely safe and does result in some performance enhancements.

Files

\$HOME/.mh_profile	The user profile
or \$MH	Rather than the standard profile
<mh-dir>/context	The user context
or \$CONTEXT	Rather than the standard context
<folder>/.mh_sequences	Public sequences for <folder>

Profile Components

All

See Also

mh(1), environ(5)

Defaults

None

Context

All

History

In previous versions of *MH*, the current-message value of a writable folder was kept in a file called “cur” in the folder itself. In *mh.3*, the *.mh_profile* contained the current-message values for all folders, regardless of their writability.

In all versions of *MH* since *mh.4*, the *.mh_profile* contains only static information, which *MH* programs will **NOT** update. Changes in context are made to the *context* file kept in the users *MH directory*. This includes, but is not limited to: the “Current-Folder” entry and all private sequence information. Public sequence information is kept in a file called *.mh_sequences* in each folder.

To convert from the format used in releases of *MH* prior to the format used in the *mh.4* release, *install-mh* should be invoked with the “-compat” switch. This generally happens automatically on *MH* systems generated with the “COMPAT” option during *MH* configuration.

The *.mh_profile* may override the path of the *context* file, by specifying a “context” entry (this must be in lower-case). If the entry is not absolute (does not start with a /), then it is interpreted relative to the user’s *MH directory*. As a result, you can actually have more than one set of private sequences by using different context files.

NAME

ap – parse addresses 822-style

SYNOPSIS

```
/usr/new/lib/mh/ap [-form formatfile] [-format string] [--normalize] [--nonnormalize]
[-width columns] addr ... [-help]
```

DESCRIPTION

ap is a program that parses addresses according to the ARPA Internet standard. It also understands many non-standard formats. It is useful for seeing how *MH* will interpret an address.

The *ap* program treats each argument as one or more addresses, and prints those addresses out in the official 822-format. Hence, it is usually best to enclose each argument in double-quotes for the shell.

To override the output format used by *ap*, the ‘--format string’ or ‘--format file’ switches are used. This permits individual fields of the address to be extracted with ease. The string is simply a format string and the file is simply a format file. See *mh-format* (5) for the details.

In addition to the standard escapes, *scan* also recognizes the following additional escape:

escape substitution

error a diagnostic if the parse failed

If the ‘--normalize’ switch is given, *ap* will try to track down the official hostname of the address.

Here is the default format string used by *ap*:

```
%<{error}%{error}: %{text}%|%(putstr(proper(text)))%>
```

which says that if an error was detected, print the error, a ‘:’, and the address in error. Otherwise, output the 822-proper format of the address.

Files

<code>\$HOME/.mh_profile</code>	The user profile
<code>/usr/new/lib/mh/mtstailor</code>	tailor file

Profile Components

None

See Also

dp(8),
Standard for the Format of ARPA Internet Text Messages (aka RFC-822)

Defaults

‘--format’ defaults as described above
‘--normalize’
‘--width’ defaults to the width of the terminal

Context

None

NAME

conflict – search for alias/password conflicts

SYNOPSIS

/usr/new/lib/mh/conflict [-mail name] [-search directory] [aliasfiles...] [--help]

DESCRIPTION

Conflict is a program that checks to see if the interface between *MH* and transport system is in good shape

Conflict also checks for maildrops in /usr/spool/mail which do not belong to a valid user. It assumes that no user name will start with '.', and thus ignores files in /usr/spool/mail which begin with '.'. It also checks for entries in the *group* (5) file which do not belong to a valid user, and for users who do not have a valid group number. In addition duplicate users and groups are noted.

If the '-mail name' switch is used, then the results will be sent to the specified *name*. Otherwise, the results are sent to the standard output.

The '-search directory' switch can be used to search directories other than /usr/spool/mail and to report anomalies in those directories. The '-search directory' switch can appear more than one time in an invocation to *conflict*.

Conflict should be run under *cron* (8), or whenever system accounting takes place.

Files

/usr/new/lib/mh/mtstailor	tailor file
/etc/passwd	List of users
/etc/group	List of groups
/usr/new/mh/mhmail	Program to send mail
/usr/spool/mail/	Directory of mail drop

Profile Components

None

See Also

mh-alias(5)

Defaults

'aliasfiles' defaults to /usr/new/lib/mh/MailAliases

Context

None

NAME

install-mh – initialize the MH environment

SYNOPSIS

/usr/new/lib/mh/install-mh [-auto] [-compat]

DESCRIPTION

When a user runs any *MH* program for the first time, the program will invoke *install-mh* (with the ‘-auto’ switch) to query the user for the initial *MH* environment. The user does **NOT** invoke this program directly. The user is asked for the name of the directory that will be designated as the user’s *MH* directory. If this directory does not exist, the user is asked if it should be created. Normally, this directory should be under the user’s home directory, and has the default name of Mail/. After *install-mh* has written the initial .mh_profile for the user, control returns to the original *MH* program.

As with all *MH* commands, *install-mh* first consults the \$HOME environment variable to determine the user’s home directory. If \$HOME is not set, then the */etc/passwd* file is consulted.

When converting from *mh.3* to *mh.4*, *install-mh* is automatically invoked with the ‘-compat’ switch.

Files

\$HOME/.mh_profile	The user profile
--------------------	------------------

Profile Components

Path:	To set the user’s MH directory
-------	--------------------------------

Context

With ‘-auto’, the current folder is changed to “inbox”.

See Also

Standard for the Format of ARPA Internet Text Messages (aka RFC-822),
mhtml(1), send(1), mh-mail(5), mh-alias(5)

Defaults

'-alias /usr/new/lib/mh/MailAliases'
'-format'
'-nomsgid'
'-noverbose'
'-width 72'
'-nofilter'

Context

None

Bugs

"Reply-To:" fields are allowed to have groups in them according to the 822 specification, but *post* won't let you use them.

6. ADVANCED FEATURES

This section describes some features of *MH* that were included strictly for advanced *MH* users. These capabilities permit *MH* to exhibit more powerful behavior for the seasoned *MH* users.

USER-DEFINED SEQUENCES

User-defined sequences allow the *MH* user a tremendous amount of power in dealing with groups of messages in the same folder by allowing the user to bind a group of messages to a meaningful symbolic name. The user may choose any name for a message sequence, as long as it consists of alphanumeric characters and does not conflict with the standard *MH* reserved message names (e.g., "first", etc). After defining a sequence, it can be used wherever an *MH* command expects a 'msg' or 'msgs' argument. Although all *MH* commands expand user-defined sequences as appropriate, there are two commands that allow the user to define and manipulate them: *pick* and *mark*.

Pick and User-Defined Sequences

Most users of *MH* will use user-defined sequences only with the *pick* command. By giving the '-sequence name' switch to *pick* (which can occur more than once on the command line), each sequence named is defined as those messages which *pick* matched according to the selection criteria it was given. Hence,

```
pick -from frated -seq fred
```

finds all those messages in the current folder which were from "frated", creates a sequence called "fred", and then adds them to the sequence. The user could then invoke

```
scan fred
```

to get a *scan* listing of those messages. Note that by default, *pick* creates the named sequences before it adds the selected messages to the sequence. Hence, if the named sequence already existed, the sequence is destroyed prior to being re-defined (nothing happens to the messages that were a part of this sequence, they simply cease to be members of that sequence). By using the '-nozero' switch, this behavior can be inhibited, as in

```
pick -from frated -seq sgroup
pick -from fear -seq sgroup -nozero
pick -from freida -seq sgroup -nozero
```

finds all those messages in the current folder which were from "frated", "fear", or "freida", and defines the sequence called "sgroup" as exactly those messages. These operations amounted to an "inclusive-or" of three selection criteria, using *pick*, one can also generate the "and" of some selection criteria as well:

```
pick -from frated -seq fred
pick -before friday -seq fred fred
```

This example defines the sequence called "fred" as exactly those messages from "frated" that were

changed, “notseen” will have to be updated. Another way to achieve this is to define the entry “Sequence–Negation:” in the `.mh_profile`. If the entry was

```
Sequence–Negation: not
```

then anytime an *MH* command was given “notseen” as a ‘msg’ or ‘msgs’ argument, it would substitute all messages that are not a member of the sequence “seen”. That is,

```
refile notseen +new
```

does just that. The value of the “Sequence–Negation:” entry in the profile can be any string. Hence, experienced users of *MH* do not use a word, but rather a special character which their shell does not interpret (users of the *CShell* use a single caret or circumflex (usually shift–6), while users of the Bourne shell use an exclamation–mark). This is because there is nothing to prevent a user of *MH* from defining a sequence with this string as its prefix, if the string is nothing by letters and digits. Obviously, this could lead to confusing behavior if the “Sequence–Negation:” entry leads *MH* to believe that two sequences are opposites by virtue of their names differing by the prefix string.

The Previous Sequence

Many times users find themselves issuing a series of commands on the same sequences of messages. If the user first defined these messages as a sequence, then considerable typing may be saved. If the user doesn’t have this foresight, *MH* provides a handy way of having *MH* remember the ‘msgs’ or ‘msg’ argument last given to an *MH* command. If the entry “Previous–Sequence:” is defined in the `.mh_profile`, then when the command finishes, it will define the sequence(s) named in the value of this entry as being exactly those messages that were specified. Hence, a profile entry of

```
Previous–Sequence: pseq
```

directs any *MH* command that accepts a ‘msg’ or ‘msgs’ argument to define the sequence “pseq” as those messages when it finishes. More than one sequence name may be placed in this entry, separated with spaces. The one disadvantage of this approach is that the *MH* programs have to update the sequence information for the folder each time they run (although most programs read this information, usually only *pick* and *mark* have to write this information out).

The Unseen Sequence

Finally, some users like to distinguish between messages which have been previously seen by them. Both *inc* and *show* honor the profile entry “Unseen–Sequence” to support this activity. Whenever *inc* places new messages in a folder, if the entry “Unseen–Sequence” is defined in the `.mh_profile`, then when the command finishes, *inc* will add the new messages to the sequence(s) named in the value of this entry. Hence, a profile entry of

```
Unseen–Sequence: unseen
```

directs *inc* to add new messages to the sequence “unseen”. Unlike the behavior of the “Previous–Sequence” entry in the profile however, the sequence(s) will not be zero’d.

Similarly, whenever *show* (or *next* or *prev*) displays a message, they remove those messages from any sequences named by the “Unseen–Sequence” entry in the profile.

COMPOSITION OF MAIL

There are a number of interesting advanced facilities for the composition of outgoing mail.

```
sendf
```

Or, if more editing was required, the draft could be edited with

```
comp -use
```

Instead, if other drafts had been composed in the meantime, so that this message draft was no longer known as 'cur' in the 'draft' folder, then the user could *scan* the folder to see which message draft in the folder should be used for editing or sending. Clever users could even employ a back-quoted *pick* to do the work:

```
comp -use 'pick +drafts -to bug-mh'
```

or

```
sendf 'pick +drafts -to bug-mh'
```

Note that in the *comp* example, the output from *pick* must resolve to a single message draft (it makes no sense to talk about composing two or more drafts with one invocation of *comp*). In contrast, in the *send* example, as many message drafts as desired can appear, since *send* doesn't mind sending more than one draft at a time.

Note that the argument '-draftfolder +folder' is not included in the profile entry for *send*, since when *comp*, et. al., invoke *send* directly, they supply *send* with the UNIX pathname of the message draft, and not a 'draftmessage msg' argument. As far as *send* is concerned, a *draft folder* is not being used.

It is important to realize that *MH* treats the draft folder like a standard *MH* folder in nearly all respects. There are two exceptions: first, under no circumstances will the '-draftfolder folder' switch cause the named folder to become the current folder.³ Second, although conceptually *send* deletes the 'msgs' named in the draft folder, it does not call 'delete-prog' to perform the deletion.

What Happens if the Draft Exists

When the *comp*, *dist*, *forw*, and *repl* commands are invoked and the draft you indicated already exists, these programs will prompt the user for a response directing the program's action. The prompt is

```
Draft "/usr/src/uci/mh/mhbox/draft" exists (xx bytes).
Disposition?
```

The appropriate responses and their meanings are: replace: deletes the draft and starts afresh; list: lists the draft; refile: files the draft into a folder and starts afresh; and, quit: leaves the draft intact and exits. In addition, if you specified '-draftfolder folder' to the command, then one other response will be accepted: new: finds a new draft, just as if '-draftmessage new' had been given. Finally, the *comp* command will accept one more response: use: re-uses the draft, just as if '-use' had been given.

³ Obviously, if the folder appeared in the context of a standard '+folder' argument to an *MH* program, as in

```
scan +drafts
```

it might become the current folder, depending on the context changes of the *MH* program in question.

increment its value) to use as the volume number.

Having calculated the name of the digest and the volume and issue numbers, *forw* will now process the components file using the same format string mechanism used by *repl*. The current ‘%’-escapes are:

<i>escape type</i>	<i>substitution</i>
digest string	digest name
issue integer	issue number
volume integer	volume number

In addition, to capture the current date, any of the escapes valid for *dp* (8) are also valid for *forw*.

The default components file used by *forw* when in digest mode is:

```
\` .so /usr/new/lib/mh/digestcomps included inline here so it looks good
From:  %(digest)-Request
To:    %(digest) Distribution: dist-%(digest);
Subject: %(digest) Digest V%(putnum(msg)) #%(putnum(cur))
Reply-To: %(digest)
-----
%(digest) Digest          %(putstr(weekday(date))), %2(putnumf(mday(date))) \
%(putstr(month(date))) 19%02(putnumf(year(date)))
                          Volume %(putnum(msg)) : Issue %(putnum(cur))
```

Today's Topics:

Hence, when the ‘-digest’ switch is present, the first step taken by *forw* is to expand the format strings in the component file. The next step is to compose the draft using the standard digest encapsulation algorithm (even putting an “End of list Digest” trailer in the draft). Once the draft is composed by *forw*, *forw* writes out the volume and issue profile entries for the digest, and then invokes the editor.

Naturally, when composing the draft, *forw* will honor the ‘-filter filterfile’ switch, which is given to *mhl* to filter each message being forwarded prior to encapsulation in the draft. A good filter file to use, which is called *mhl.digest*, is:

```
width=80,overflowoffset=10
leftadjust,compress,compwidth=9
Date:formatfield="%<(nodate(text))%(text)|%(putstr(tws(text)))%>"
From:
Subject:
:
body:nocomponent,overflowoffset=0,noleftadjust,nocompress
```

FOLDER HANDLING

There are two interesting facilities for manipulating folders: relative folder addressing, which allows a user to shorten the typing of long folder names; and the folder-stack, which permits a user to keep a stack of current folders.

Appendix A
COMMAND SUMMARY

ali [-alias aliasfile] [-list] [-nolist] [-normalize] [-nonormalize] [-user] [-nouser] names ...
 [-help]

anno [+folder] [msgs] [-component field] [-inplace] [-noinplace] [-text body] [-help]

burst [+folder] [msgs] [-inplace] [-noinplace] [-quiet] [-noquiet] [-verbose] [-noverbose]
 [-help]

comp [+folder] [msg] [-draftfolder +folder] [-draftmessage msg] [-nodraftfolder]
 [-editor editor] [-noedit] [-file file] [-form formfile] [-use] [-nouse]
 [-whatnowproc program] [-nowhatnowproc] [-help]

dist [+folder] [msg] [-annotate] [-noannotate] [-draftfolder +folder] [-draftmessage msg]
 [-nodraftfolder] [-editor editor] [-noedit] [-form formfile] [-inplace] [-noinplace]
 [-whatnowproc program] [-nowhatnowproc] [-help]

folder [+folder] [msg] [-all] [-fast] [-nofast] [-header] [-noheader] [-pack] [-nopack]
 [-recurse] [-norecurse] [-total] [-nototal] [-print] [-noprint] [-list] [-nolist] [-push]
 [-pop] [-help]

folders

forw [+folder] [msgs] [-annotate] [-noannotate] [-draftfolder +folder] [-draftmessage msg]
 [-nodraftfolder] [-editor editor] [-noedit] [-filter filterfile] [-form formfile] [-format]
 [-noformat] [-inplace] [-noinplace] [-whatnowproc program] [-nowhatnowproc]
 [-help]

forw [+folder] [msgs] [-digest list] [-issue number] [-volume number]
 [other switches for *forw*] [-help]

inc [+folder] [-audit audit-file] [-noaudit] [-change cur] [-nochange cur] [-file name]
 [-form formatfile] [-format string] [-silent] [-nosilent] [-truncate] [-notruncate]
 [-width columns] [-help]

mark [+folder] [msgs] [-sequence name ...] [-add] [-delete] [-list] [-public] [-npublic]
 [-zero] [-nozero] [-help]

scan [+folder] [msgs] [--clear] [--noclear] [--form formatfile] [--format string] [--header] [--noheader] [--width columns] [--help]

send [--alias aliasfile] [--draft] [--draftfolder +folder] [--draftmessage msg] [--nodraftfolder] [--filter filterfile] [--nofilter] [--format] [--noformat] [--forward] [--noforward] [--msgid] [--nomsgid] [--push] [--nopush] [--verbose] [--noverbose] [--watch] [--nowatch] [--width columns] [file ...] [--help]

show [+folder] [msgs] [--draft] [--header] [--noheader] [--showproc program] [--noshowproc] [switches for *showproc*] [--help]

sortm [+folder] [msgs] [--datefield field] [--verbose] [--noverbose] [--help]

vmh [--prompt string] [--vmhproc program] [--novmhproc] [switches for *vmhproc*] [--help]

whatnow [--draftfolder +folder] [--draftmessage msg] [--nodraftfolder] [--editor editor] [--noedit] [--prompt string] [file] [--help]

whom [--alias aliasfile] [--check] [--nocheck] [--draft] [--draftfolder +folder] [--draftmessage msg] [--nodraftfolder] [file] [--help]

/usr/new/lib/mh/ap [--form formatfile] [--format string] [--normalize] [--nonormalize] [--width columns] [addr ...] [--help]

/usr/new/lib/mh/conflict [--mail name] [--search directory] [aliasfiles ...] [--help]

/usr/new/lib/mh/dp [--form formatfile] [--format string] [--width columns] [dates ...] [--help]

/usr/new/lib/mh/install-mh [--auto] [--compat]

/usr/new/lib/mh/post [--alias aliasfile] [--filter filterfile] [--nofilter] [--format] [--noformat] [--msgid] [--nomsgid] [--verbose] [--noverbose] [--watch] [--nowatch] [--width columns] [file] [--help]

REFERENCES

1. Crocker, D. H., J. J. Vittal, K. T. Pogran, and D. A. Henderson, Jr., "Standard for the Format of ARPA Network Text Messages," *RFC733*, November 1977.
2. Thompson, K., and D. M. Ritchie, "The UNIX Time-sharing System," *Communications of the ACM*, Vol. 17, July 1974, pp. 365-375.
3. McCauley, E. J., and P. J. Drongowski, "KSOS—The Design of a Secure Operating System." *AFIPS Conference Proceedings*, National Computer Conference, Vol. 48, 1979, pp. 345-353.
4. Crocker, David H., *Framework and Functions of the "MS" Personal Message System*, The Rand Corporation, R-2134-ARPA, December 1977.
5. Thompson, K., and D. M. Ritchie, *UNIX Programmer's Manual*, 6th ed., Western Electric Company, May 1975 (available only to UNIX licensees).
6. Crocker, D. H., "Standard for the Format of ARPA Internet Text Messages," *RFC822*, August 1982.

How to Read the Network News

Mark R. Horton
AT&T Bell Laboratories
Columbus, OH 43213

Revised by Rick Adams for 2.10.3

What is the Network News?

USENET (Users' Network) is a bulletin board shared among many computer systems around the world. USENET is a logical network, sitting on top of several physical networks, among them *UUCP*, *BLICN*, *BERKNET*, *X.25*, and the *ARPANET*. Sites on USENET include many universities, private companies and research organizations. Most of the members of USENET are either university computer science departments or part of AT&T. Currently, there are over 2000 USENET sites in the USA, Canada, Europe, Japan and Korea with more joining every day. Most are running the UNIX† operating system.

The network news, or simply *netnews*, is the set of programs that provide access to the news and transfer it from one machine to the next. Netnews was originally written at Duke University and has been modified extensively by the University of California at Berkeley and others. Netnews allows articles to be posted for limited or very wide distribution. This document contains a list of newsgroups that were active at the time the document was written. It exists to assist you in determining which newsgroups you may want to subscribe to. When creating a new article, the level of distribution can be controlled by use of the "Distribution" field. This will prevent notices of apartments for rent in New Jersey being broadcast to California (or even Europe).

Any user can post an article, which will be sent out to the network to be read by persons interested in that topic. You can specify which topics are of interest to you by putting them in a *subscription list*. Then, whenever you ask to read news, the news reading program will present all unread articles of interest. There are also facilities for browsing through old news, posting follow-up articles, and sending direct electronic mail replies to the author of an article.

This paper is a tutorial, aimed at the user who wants to read and possibly post news. The system administrator who must install the software should see the companion document *USENET Version B Installation*.

Why USENET?

USENET is useful in a number of ways. Someone wishing to announce a new program or product can reach a wide audience. A user can ask "Does anyone have an *x*?" and will usually get several responses within a day or two. Bug reports and their fixes can be made quickly available without the usual overhead of sending out mass mailings. Discussions involving many people at different locations can take place without having to get everyone together.

Another facility with similar capabilities to *netnews* is the *electronic mailing list*. A mailing list is a collection of electronic mailing addresses of users who are interested in a particular topic. By

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

article.

Among the other commands you can type after seeing the header of an article are:

- x** Exit *readnews*. This is different from **q** in that the **q** command will update the record of which articles you have read, but **x** will pretend you never started *readnews*.
- N** Go on to the next newsgroup. The remaining articles in the current newsgroup are considered *unread*, and will be offered to you again the next time you read news.
- s file** The article is saved in a disk file with the given name. In practice, what usually happens is that an article is printed, and then *readnews* goes on to print the header of the next article before you get a chance to type anything. So you usually want to write out the *previous* message (the last one you have read in full); in this case, use the form *s-filename*.
- e** Erase the memory of having seen this article. It will be offered to you again next time, as though you had never seen it. The **e-** case variation (erase memory of the previously read article instead of the current article) is useful for checking follow-ups to see if anyone has already said what you wanted to say.
- r** Reply to the author of the message. You will be placed in the editor, with a set of headers derived from the message you are replying to. Type in your message after the blank line. If you wish to edit the header list to add more recipients or send carbon copies, for instance, you can edit the header lines. Anyone listed on a line beginning with "To" or "Cc" will receive a copy of your reply. Note that the path used to receive a piece of news may not be the fastest way to reply by mail. If speed is important and you know a faster way, edit it in place of what the reply command supplied. A mail command will then be started up, addressed to the persons listed in the header. You are then returned to *readnews*. The case **r-** is also useful to reply to the previous message. Another variation on this is **rd-** which puts you in \$MAILER (or *mail(1)* by default) to type in your reply directly.
- f** Post a follow-up message to the same newsgroup. This posts an article on this newsgroup with the same title as the original article. Use common sense when posting follow-ups. (Read Matt Bishop's paper "How to use USENET Effectively" for extended discussion of when and when not to post -- many follow-up articles should have just been replies.) You will be placed in the editor. Enter your message and exit. The case **f-** is also useful to follow up the previous message. In each case, the editor you are placed in will be *vi(1)* unless you set **EDITOR** (in your environment) to some other editor. You should enter the text of the follow-up after the blank line.
- +** The article is skipped for now. The next time you read news, you will be offered this article again.
- Go back to the previous article. This toggles, so that two **-**'s get you the current article.
- b** Back up one article in the current group. This is not necessarily the previous article.
- U** Unsubscribe from this newsgroup. Your *.newsrct(5)* file will be edited to change the : for that newsgroup to an ! preventing you from being shown that newsgroup again.
- ?** If you type any unrecognized command, a summary of valid commands will be printed.

Changing your Subscription List

If you take no special action you will subscribe to a default subscription list. This default varies locally. To find out your local default, type

```
readnews -s
```

Typically this list will include all newsgroups ending in "general", such as **general**, and **net.general**. (As distributed, the default is **general.all.general**. Another popular default is **all**.) You can change this by creating a file in your home directory named *.newsrct* which contains as its first line a line of the

about the headers while you are still in the editor, you can edit them as well. Extra headers can also be added before the blank line.

Browsing through Old News

There are a number of command line options to the *readnews* command to help you find an old article you want to see again. The *-n newsgroups* option restricts your search to certain newsgroups. The *-x* option arranges to ignore the record of articles read, which is kept in your *.newsrc* file. This will cause all articles in all newsgroups to which you subscribe to be displayed, even those which you have already seen. It also causes *readnews* to not update the *.newsrc* file. The *-a date* option asks for news received after the given *date*. Note that even with the *-a* option, only articles you have not already seen will be printed, unless you combine it with the *-x* option. (Articles are kept on file until they expire, typically after two weeks.) The *-t keywords* option restricts the query to articles mentioning one of the *keywords* in the title of the article. Thus, the command

```
readnews -n net.unix -x -a last thursday -t setuid
```

asks for all articles in newsgroup *net.unix* since last Thursday about the *setuid* feature. (Be careful with the *-t* option. The above example will not find articles about "suid", nor will it find articles with no title or whose author did not use the word "setuid" in the title.)

Other useful options include the *-l* option (which lists only the headers of articles - a useful form for browsing through lots of messages.) The *-p* option prints the messages without asking for any input; this is similar to some older news programs on many UNIX systems and is useful for directing output to a printer. The *-r* option produces articles in reverse order, from newest to oldest.

User Interfaces

The *user interface* of a program is the view it presents to the user, that is, what it prints and what it allows you to type. *Readnews* has options allowing you to use different user interfaces. The interface described above is called the "msgs" interface because it mimics the style of the Berkeley *msgs(1)* program. (This program, in turn, mimics a program at MIT of the same name.) The key element of the *msgs* interface is that after printing the header, you are asked if you want the rest of the message.

Another interface is available with the *-c* option. In this case, the entire message is printed, header and body, and you are prompted at the end of the message. The command options are the same as the *msgs* interface, but it is usually not necessary to use the *-* suffix on the *r*, *s*, or *f* commands. This interface is called the "/bin/mail" (pronounced "bin mail") interface, because it mimics the UNIX program of that name.

A third interface is the *Mail(1)* (pronounced "cap mail") interface, available with the *-M* option. This invokes the *Mail* program directly, and allows you to read news with the same commands as you read mail. (This interface may not work on your system - it requires a special version of *Mail* with a *-T* option.)

A fourth interface, is the MH news/mail program from Rand. That program can be used directly to read network news.

A fifth interface, *vnews*, which works well on display terminals, is described in the Appendix.

A sixth possibility is the *notesfile* system, described in a separate paper. It is also display-oriented.

A seventh possibility is to use your favorite mail system as an interface. There are a number of different mail reading programs, including */bin/mail(1)*, *Mail*, *msg(1)*, and MH. Any mail system with an option to specify an alternative mailbox can be used to read news. For example, to use *Mail* without the *-M* option, type

```
readnews -c "Mail -f %"
```

The shell command in quotes is invoked as a child of *readnews*. The *-f* option to *Mail* names the alternative mailbox. *Readnews* will put the news in a temporary file, and give the name of this file to

List of Newsgroups

This section lists the newsgroups that are currently active. It is intended to help you decide what you want to subscribe to. Note that the list is constantly changing. Note also that this list only describes those groups available on a network-wide basis. Since not all installations choose to receive all newsgroups, it is recommended that each installation edit the list of local newsgroups to be correct before distributing this document to their users. If this is not possible, a local appendix can be created.

Local

Local groups are kept on the current machine only. Local names can be identified by the lack of a prefix, that is, there are no periods in local newsgroup names.

general News to be read by everyone on the local machine. For example: "The system will be down Monday morning for PM." Or, "A new version of program *x* has been installed." This newsgroup is usually mandatory – you are required to subscribe to this newsgroup. (The list of mandatory newsgroups varies locally.) This requirement assures that important announcements reach all users. (Formerly *msgs*.)

Network Wide

These are the groups as of the last editing of this manual. The list is undoubtedly already out of date. A current list can be obtained by typing ? to the "Newsgroups?" prompt in postnews.

net.abortion	All sorts of discussions on abortion.
net.ai	Artificial intelligence discussions.
net.analog	Analog design developments, ideas, and components.
net.announce	Moderated, general announcements of interest to all.
net.announce.newusers	Moderated, explanatory postings for new users.
net.announce.arpa-internet	Announcements from the Arpa world.
net.arch	Computer architecture.
net.astro	Astronomy discussions and information.
net.astro.expert	Discussion by experts in astronomy.
net.audio	High fidelity audio.
net.auto	Automobiles, automotive products and laws.
net.auto.tech	Technical aspects of automobiles, et. al.
net.aviation	Aviation rules, means, and methods.
net.bicycle	Bicycles, related products and laws.
net.bio	Biology and related sciences.
net.books	Books of all <i>genres</i> , shapes, and sizes.
net.bugs	General bug reports and fixes.
net.bugs.2bsd	Reports of UNIX* version 2BSD related bugs.
net.bugs.4bsd	Reports of UNIX version 4BSD related bugs.
net.bugs.usg	Reports of USG (System III, V, etc.) bugs.
net.bugs.uucp	Reports of UUCP related bugs.
net.bugs.v7	Reports of UNIX V7 related bugs.
net.cog-eng	Cognitive engineering.
net.college	College, college activities, campus life, etc.
net.columbia	The space shuttle and the STS program.
net.comics	The funnies, old and new.
net.consumers	Consumer interests, product reviews, etc.
net.cooks	Food, cooking, cookbooks, and recipes.
net.crypt	Different methods of data en/decryption.
net.cse	Computer science education.
net.cycle	Motorcycles and related products and laws.
net.database	Database and data management issues and theory.

net.micro.apple	Discussion about Apple micros.
net.micro.amiga	Talk about the new Amiga micro.
net.micro.atari	Discussion about Atari micros.
net.micro.att	Discussions about AT&T microcomputers
net.micro.cbm	Discussion about Commodore micros.
net.micro.cpm	Discussion about the CP/M operating system.
net.micro.hp	Discussion about Hewlett/Packard's.
net.micro.mac	Material about the Apple Macintosh & Lisa.
net.micro.pc	Discussion about IBM personal computers.
net.micro.ti	Discussion about Texas Instruments.
net.micro.trs-80	Discussion about TRS-80's.
net.misc	Various discussions too short-lived for other groups.
net.motss	Issues pertaining to homosexuality.
net.movies	Reviews and discussions of movies.
net.music	Music lovers' group.
net.music.classical	Discussion about classical music.
net.music.folk	Folks discussing folk music of various sorts.
net.music.gdead	A group for (Grateful) Dead-heads.
net.music.synth	Synthesizers and computer music.
net.net-people	Announcements, requests, etc. about people on the net.
net.news	Discussions of USENET itself.
net.news.adm	Comments directed to news administrators.
net.news.b	Discussion about B news software.
net.news.config	Postings of system down times and interruptions.
net.news.group	Discussions and lists of newsgroups
net.news.newsite	Postings of new site announcements.
net.news.notes	Notesfile software from the Univ. of Illinois.
net.news.sa	Comments directed to system administrators.
net.news.stargate	Discussion about satellite transmission of news.
net.nlang	Natural languages, cultures, heritages, etc.
net.nlang.africa	Discussions about Africa & things African.
net.nlang.celts	Group about Celts.
net.nlang.greek	Group about Greeks.
net.nlang.india	Group for discussion about India & things Indian
net.origins	Evolution versus creationism (sometimes hot!).
net.periphs	Peripheral devices.
net.pets	Pets, pet care, and household animals in general.
net.philosophy	Philosophical discussions.
net.physics	Physical laws, properties, etc.
net.poems	For the posting of poems.
net.politics	Political discussions. Could get hot.
net.politics.theory	Theory of politics and political systems.
net.puzzle	Puzzles, problems, and quizzes.
net.railroad	Real and model train fans' newsgroup.
net.rec	Recreational/participant sports.
net.rec.birds	Hobbyists interested in bird watching.
net.rec.boat	Hobbyists interested in boating.
net.rec.bridge	Hobbyists interested in bridge.
net.rec.nude	Hobbyists interested in naturist/nudist activities.
net.rec.photo	Hobbyists interested in photography.
net.rec.scuba	Hobbyists interested in SCUBA diving.
net.rec.ski	Hobbyists interested in skiing.
net.rec.skydive	Hobbyists interested in skydiving.

mod.computers.workstations	Various workstation-type computers.
mod.graphics	Graphics software, hardware, theory, etc.
mod.human-nets	Computer aided communications digest.
mod.legal	Discussions of computers and the law.
mod.map	Various maps, including UUCP maps.
mod.motss	Moderated newsgroup on gay issues and topics.
mod.movies	Moderated reviews and discussion of movies.
mod.music	Moderated reviews and discussion of things musical.
mod.newprod	Announcements of new products of interest to readers.
mod.newslists	Postings of news-related statistics and lists.
mod.os	Disussions about operating systems and related areas.
mod.os.os9	Discussions about the os9 operating system.
mod.os.unix	Moderated discussion of Unix* features and bugs.
mod.politics	Discussions on political problems, systems, solutions.
mod.politics.arms-d	Arms discussion digest.
mod.protocols	Various forms and types of FTP protocol discussions.
mod.protocols.appletalk	Applebus hardware & software discussion.
mod.protocols.kermit	Information about the Kermit package.
mod.protocols.tcp-ip	TCP and IP network protocols.
mod.rec	Discussions on pastimes (not currently active).
mod.rec.guns	Discussions about firearms.
mod.recipes	A "distributed cookbook" of screened recipes.
mod.risks	Risks to the public from computers & users.
mod.sources	Moderated postings of public-domain sources.
mod.sources.doc	Archived public-domain documentation.
mod.std	Moderated discussion about various standards.
mod.std.c	Discussion about C language standards.
mod.std.mumps	Discussion for the X11.1 committee on Mumps.
mod.std.unix	Discussion for the P1003 committee on Unix.
mod.techreports	Announcements and lists of technical reports.
mod.telecom	Telecommunications digest.
mod.test	Testing of moderated newsgroups -- no moderator.
mod.vlsi	Very large scale integrated circuits.

<CONTROL-N>

Go forwards *count* lines.

<CONTROL-Z>

Go backwards *count* lines.

<CONTROL-L>

Redraw the screen. **<CONTROL-L>** may be typed at any time.

- b** Back up one article in the current group.
- l** Redisplay the article after you have sent a follow-up or reply.
- n** Move on to the next item in a digest. "." is equivalent to **n**. This is convenient if your terminal has a keypad.
- p** Show the parent article (the article that the current article is a follow-up to). This doesn't work if the current article was posted by A-news or notesfiles. To switch between the current and parent articles, use the - command. Unfortunately, if you use several **p** commands to trace the discussion back further, there is no command to return to the original level.
- ug** Unsubscribe to the current group. This is a two character command to ensure that it is not typed accidentally and to leave room for other types of unsubscribes (*e.g.*, unsubscribe to discussion).
- v** Print the current version of the news software.
- D** Decrypts a joke. It only handles *rot13* jokes. The **D** command is a toggle; typing another **D** re-encrypts the joke.



How to Use USENET Effectively

Matt Bishop
Research Institute for Advanced Computer Science
Mail Stop 230-5
NASA Ames Research Center
Moffett Field, CA 94035

1. Introduction

USENET is a worldwide bulletin board system in which thousands of computers pass articles back and forth. Of necessity, customs have sprung up enabling very diverse people and groups to communicate peaceably and effectively using USENET. These customs are for the most part written, but are scattered over several documents that can be difficult to find; in any case, even if a new user can find all the documents, he most likely will have neither the time nor the inclination to read them all. This document is intended to collect all these conventions into one place, thereby making it easy for new users to learn about the world of USENET. (Old-timers, too, will benefit from reading this.)

You should read this document and understand it thoroughly before you even think about posting anything. If you have questions, please ask your USENET administrator (who can usually be reached by sending mail to *usenet*) or a more knowledgeable USENET user. Believe me, you will save yourself a lot of grief.

The mechanics of posting an article to USENET are explained in Mark Horton's excellent paper *How to Read the Network News*; if you have not read that yet, stop here and do so. A lot of what follows depends on your knowing (at least vaguely) the mechanics of posting news.

Before we discuss these customs, we ought to look at the history of USENET, what it is today, and why we need these conventions.

2. All About USENET

USENET began on a set of computers in North Carolina's Research Triangle. The programs involved (known as "netnews" then, and "A news" now) exchanged messages; it was a small, multi-computer bulletin board system. As time passed, administrators of other systems began to connect their computers to this bulletin board system. The network grew. Then, at Berkeley, the news programs were rewritten (this version became known as "B news") and the format changed to conform to ARPA standards (again, this became the "B protocol for news").[†] This version of news was very widely distributed, and at this point USENET began to take on its current shape.

USENET is a *logical* network (as opposed to a *physical* network.) It is also a very amorphous network, in that there is no central administration or controlling site. There is not even an official list of members, although there is a very complete unofficial one. A site gets access to USENET by finding some other site already on USENET that it can connect to and exchange news articles. So long as this second site (called a *neighbor* of the first site) remains willing and able to pass articles to and from the first site, the first site is on USENET. A site leaves the USENET only when no one is willing or able to pass articles to, or accept articles from, it.

[†] See *Standard for Interchange of USENET Messages* for a description of the two formats.



that many copies of the same answer to a simple question are posted.

If you want to repost something because you believe it did not get to other USENET sites due to transmission problems (this happens sometimes, but a lot less often than commonly believed), do some checking before you repost. If you have a friend at another USENET site, call him and ask if the article made it to his site. Ask your USENET administrator if he knows of any problems in the USENET; there are special newsgroups to which USENET administrators subscribe in which problems are reported, or he can contact his counterparts at other sites for information. Finally, if you decide you must repost it, indicate in the article subject that it is a reposting, and say why you are reposting it (if you don't, you'll undoubtedly get some very nasty mail.)

Reposting announcements of products or services is flatly forbidden. Doing so may convince other sites to turn off your USENET access.

When school starts, hoards of new users descend upon the USENET asking questions. Many of these questions have been asked, and answered, literally thousands of times since USENET began. The most common of these questions, and their answers, have been collected in the hope that the new users will read them and not re-post the same questions. So, if you want to ask a question, check Appendix I (**Answers to Frequently Asked Questions**) to be sure it isn't one that has been asked and answered literally hundreds of times before you started reading the USENET.

3.2. Do not post anything when upset, angry, or intoxicated

Posting an article is a lot like driving a car – you have to be in control of yourself. Postings which begin “Jane, you ignorant slut, ...” are very definitely considered in poor taste†. Unfortunately, they are also far too common.

The psychology of this is interesting. One popular belief is that since we interact with USENET via computers, we all often forget that a computer did not do the posting; a human did. A contributing factor is that you don't have to look the target of abuse in the eye when you post an abusive message; eye-to-eye contact has an amazing effect on inhibiting obnoxious behavior. As a result, discussions on the USENET often degenerate into a catfight far more readily than would a face-to-face discussion.

Before you post an article, think a minute; decide whether or not you are upset, angry, or high. If you are, wait until you calm down (or come down) before deciding to post something. Then think about whether or not you really want to post it. You will be amazed what waiting a day or even a few hours can do for your perspective.

Bear in mind that shouting hasn't convinced anyone of anything since the days of Charlemagne, and being abusive makes people hold even more tenaciously to their ideas or opinions. Gentleness, courtesy, and eloquence are far more persuasive; not only do they indicate you have enough confidence in your words to allow them to speak for you, but also they indicate a respect for your audience. This in turn makes it easier for your audience to like or respect you – and people tend to be far more interested in, and receptive to, arguments advanced by those they like or respect than by writers who are abusive. Finally, remember that some discussions or situations simply cannot be resolved. Because people are different, agreed-upon facts often lead to wildly different feelings and conclusions. These differences are what makes life so wonderful; were we all alike, the world would be a very boring place. So, don't get frantic; relax and enjoy the discussion. Who knows, you might even learn something!

3.3. Be sure your posting is appropriate to USENET

Some things are inappropriate to post to USENET. Discussing whether or not some other discussion is appropriate, or if it is in the right newsgroup, is an example. Invariably, the “meta-discussion” generates so many articles that the discussion is simply overwhelmed and vanishes; but

† Unless you are critiquing *Saturday Night Live*.

Part of the price of freedom is allowing others to make fools of themselves. You wouldn't like to be censored, so don't advocate censorship of others. No one is forcing you to read the postings.

In some countries, posting or receiving certain types of articles may be a criminal offense. As a result, certain newsgroups which circulate freely within the United States may not be circulated in other nations without risking civil or criminal liabilities. In this case, the appropriate action for sites in that country is neither to accept nor to transmit the newsgroup. No site is *ever* forced to accept or pass on *any* newsgroup.

4. Where to Post

The various newsgroups and distributions have various rules associated with their use. This section will describe these rules and offer suggestions on which newsgroups to post your message.

4.1. Keep the distribution as limited as possible

A basic principle of posting is to keep the distribution of your article as limited as possible. Like our modern society, USENET is suffering from both an information glut and information pollution. It is widely believed that the USENET will cease to function unless we are able to cut down the quantity of articles. One step in this direction is not to post something to places where it will be worthless. For example, if you live in Hackensack, New Jersey, the probability of anyone in Korea wanting to buy your 1972 Toyota is about as close to zero as you can get. So confine your posting to the New Jersey area.

To do this, you can either post to a local group, or post to a net-wide group and use the *distribution* feature to limit how widely your article will go. When you give your posting program (usually *postnews(1)*) a distribution, you are (in essence) saying that machines which do not recognize that distribution should not get the article. (Think of it as a subgroup based on locality and you'll get the idea.) For example, if you are posting in the San Francisco Bay Area, and you post your article to **net.auto** but give **ba** as the distribution, the article will not be sent beyond the San Francisco Bay Area (to which the **ba** distribution is local) even though you put it in a net-wide newsgroup. Had you given the distribution as **ca** (the California distribution), your article would have been sent to all Californian sites on USENET. Had you given the distribution as **net**, your article would have been sent to all sites on USENET.

4.2. Do not post the same article twice to different groups

If you have an article that you want to post to more than one group, post to both at the same time. Newer versions of the news software will show an article only once regardless of how many newsgroups it appears in. But if you post it once to each different group, all versions of news software will show it once for each newsgroup. This angers a lot of people and wastes everybody's time.

4.3. Do not post to "mod." or "net.announce" newsgroups

You may not post directly to certain newsgroups; you cannot post to some at all. Newer versions of the news software will inform you when either of these restrictions apply, but older versions of news software will not.

The **mod.** newsgroups are *bona fide* moderated newsgroups. If you want to have the appropriate moderator post something, mail it to him. (If you do not know his address, ask your USENET administrator. In some cases, the software will automatically mail, rather than post, your article to the moderator.)

The newsgroup **net.announce** and its subgroups are moderated newsgroups designed for important announcements. It is used to post important announcements that everyone on USENET can read. (**Net.general** was meant to provide such a place, but so many inappropriate messages have been posted there that a lot of people began to unsubscribe; hence, this moderated newsgroup was set up. Very few messages are posted to it, so don't be afraid to subscribe; you will not be overwhelmed.) To



4.7. Watch out for newsgroups which have special rules about posting

Some newsgroups have special rules. This section summarizes them.

net.books	Do not post anything revealing a plot or a plot twist without putting the word "spoiler" somewhere in the "Subject" field. This will let those who do not wish to have a surprise spoiled skip the article.
net.followup	This group is for followups to articles posted in net.general or for results of surveys. No discussions are allowed.
net.jokes	If you want to post an offensive joke (this includes racial, religious, sexual, and scatological humor, among other kinds) rotate it. (If you do not know what this means, look in the section Writing Your Posting .)
net.movies	Do not post anything revealing a plot or a plot twist without putting the word "spoiler" in the "Subject" field. This will let those who do not wish to have a surprise spoiled skip the article.
net.news.group	Discussions about whether or not to create new groups, and what to name them, go here. Please mail your votes to the proposer; don't post them.
net.sources	Source code postings go here. Discussions are not allowed. Do not post bug fixes here.
net.sources.bugs	Bug reports and bug fixes to sources posted in net.sources go here.
net.test	Use the smallest distribution possible. In the body of the message, say what you are testing.
net.wanted	Requests for things other than source code go here. Please use the smallest distribution possible. Post offers here, too.
net.wanted.sources	Requests for sources go here.

5. Writing the Article

Here are some suggestions to help you communicate effectively with others on the USENET. Perhaps the best advice is not to be afraid to consult a book on writing style; two of the best are *How to Write for the World of Work* by Cunningham and Pearsall, and *Elements of Style* by Strunk and White.

5.1. Write for your audience

USENET is an international network, and any article you post will be *very* widely read. Even more importantly, your future employers may be among the readers! So, try to make a good impression.

A basic principle of all writing is to write at your readers' reading level. It is better to go below than above. Aiming where "their heads ought to be" may be fine if you are a college professor (and a lot of us would dispute even that), but it is guaranteed to cause people to ignore your article. Studies have shown that the average American reads at the fifth grade level and the average professional reads at the twelfth grade level.

5.2. Be clear and concise

Remember that you are writing for a very busy audience; your readers will not puzzle over your article. So be very clear and very concise. Be precise as well; choose the least ambiguous word you can, taking into account the context in which you are using the word. Split your posting into sections and paragraphs as appropriate. Use a descriptive title in the "Subject" field, and be sure that the title is related to the body of the article. If the title is not related, feel free to change it to a title that is.

6. Conclusion and Summary

Here is a list of the rules given above:

▀ Deciding to post

- Do not repeat postings
- Do not post anything when upset, angry, or intoxicated
- Be sure your posting is appropriate to USENET
- Do not post other people's work without permission
- Don't forget that opinions are those of the poster and not his company

▀ Where to Post

- Keep the distribution as limited as possible
- Do not post the same article twice to different groups
- Do not post to **mod.**, or **net.announce** newsgroups
- Do not post to **net.general**
- Ask someone if you can't figure out where to post your article
- Be sure there is a consensus before creating a new newsgroup
- Watch out for newsgroups which have special rules about posting

▀ Writing the Article

- Write for your audience
- Be clear and concise
- Proofread your article
- Be extra careful with announcements of products or services
- Indicate sarcasm and humor
- Mark postings which spoil surprises
- Rotate offensive postings
- The shorter your signature, the better

The USENET can be a great place for us all. Sadly, not enough people are following the customs that have been established to keep the USENET civilized. This document was written to educate all users of the USENET on their responsibilities. Let's clean up the USENET, and turn it into a friendly, helpful community again!

Acknowledgements: The writing of this document was inspired by Chuq von Rospach's posting on USENET etiquette, and it draws on previous work by Mark Horton, A. Jeff Offutt, Gene Spafford, and Chuq von Rospach.



can give it to you even if he works at a different location. If you must try the net, use newsgroup **net.net-people**, *not* **net.general**.

9. **net.math**: Proofs that $1 = 0$.

Almost everyone has seen one or more of these in high school. They are almost always based on either division by 0 or taking the square root of a negative number.

10. **net.games**: Where can I get the source for *empire(6)* or *rogue(6)*?

You can't. The authors of these games, as is their right, have chosen not to make the sources available.

11. **net.unix-wizards**: How do I remove files with non-ASCII characters in their names?

You can try to find a pattern that uniquely identifies the file. This sometimes fails because a peculiarity of some shells is that they strip off the high-order bit of characters in command lines. Next, you can try an "rm -i", or "rm -r" (see *rm(1)*). Finally, you can mess around with i-node numbers and *find(1)*.

12. **net.unix-wizards**: There is a bug in the way UNIX handles protection for programs that run *setuid*.

There are indeed problems with the treatment of protection in *setuid* programs. When this is brought up, suggestions for changes range from implementing a full capability list arrangement to new kernel calls for allowing more control over when the effective id is used and when the real id is used to control accesses. Sooner or later you can expect this to be improved. For now you just have to live with it.

13. **net.women**: What do you think about abortion?

Although abortion might appear to be an appropriate topic for **net.women**, more heat than light is generated when it is brought up. Since the newsgroup **net.abortion** has been created, all abortion-related discussion should take place there.

14. **net.singles**: What do "MOTOS," "MOTSS," "MOTAS", and "SO" stand for?

Member of the opposite sex, member of the same sex, member of the appropriate sex, and significant other, respectively.

15. **net.columbia**: Shouldn't this name be changed?

The name was devised to honor the first space shuttle. It was realized at the time the group began that the name would quickly become out of date. The intent was to create a bit of instant nostalgia.

16. **net.columbia**: Shouldn't this group be merged with **net.space**? No. **Net.columbia** is for timely news bulletins. **Net.space** is for discussions.

17. How do I use the "Distribution" feature?

When *postnews(1)* prompts you for a distribution, it's asking how widely distributed you want your article. The set of possible replies is different, depending on where you are, but at Bell Labs in Murray Hill, New Jersey, possibilities include:

```

mh3bc1 local to this machine
mh      Bell Labs, Murray Hill Branch
nj      all sites in New Jersey
btl     All Bell Labs machines
att     All AT&T machines
usa     Everywhere in the USA
na      Everywhere in North America
net     Everywhere on USENET in the world (same as "world")

```

If you hit <RETURN>, you'll get the default, which is the first part of the newsgroup name. This default is often not appropriate - *please* take a moment to think about how far away people are likely to be interested in what you have to say. Used car ads, housing wanted ads, and things for sale other than specialized equipment like computers certainly shouldn't be distributed to Europe

Report No. UIUCDCS-R-82-1081



NOTESFILE REFERENCE MANUAL
(abridged)

by

Raymond B. Essick IV
Rob Kolstad

February 14, 1983
(Revised: October 20, 1985)
(Printed: April 8, 1986)

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
1304 W. SPRINGFIELD AVENUE
URBANA, ILLINOIS 61801-2987

Supported in part by NASA Project NAS-1-138

TABLE OF CONTENTS

1 Introduction	1
2 Using Notesfiles	1
2.1 Invocation.....	1
2.2 Notesfile Names and Wildcards.....	2
2.3 The -f Option.....	3
2.4 General.....	3
2.4.1 Help.....	3
2.4.2 Exiting.....	3
2.4.3 Shells.....	4
2.4.4 Comments & Suggestions.....	4
2.5 The Index Page.....	4
2.5.1 Scrolling the Index Page.....	5
2.5.2 Choosing Notes & Responses.....	5
2.6 Notes & Responses.....	5
2.6.1 Reading Notes.....	5
2.6.2 Reading Responses.....	7
2.6.3 Writing Notes & Responses.....	7
2.6.4 Mailing Notesfile Text.....	8
2.6.5 Forwarding Text To Other Notesfiles.....	8
2.6.6 Saving Text in Local Files.....	8
2.6.7 Deletion.....	8
2.6.8 Online Communication.....	8
2.6.9 Editing Note Titles.....	9
2.6.10 Editing Notes/Responses.....	9
2.7 Other Commands.....	9
2.7.1 Returning to the Index Page.....	9
2.7.2 Searching Titles for Keywords.....	9
2.7.3 Searching for Authors.....	9
2.7.4 Stacking Notesfiles.....	9
2.7.5 Accessing Archives.....	10
2.7.6 Policy Note.....	10
2.8 The Sequencer.....	10
2.8.1 Seeing New Notes and Responses.....	10
2.8.2 Alternate Sequencers.....	11
2.8.3 Automatic Sequencing.....	11
2.9 Environment Variables.....	12
3 Other Notesfile Utilities	13
3.1 Hard Copy Output.....	13
3.2 Piped Insertion of Notes.....	13
3.3 User Subroutines.....	13
3.3.1 Nfcomment.....	13
3.3.2 Nfabort.....	14
3.4 Statistics.....	14
3.5 Checking for New Notes.....	15

APPENDICES



1 Introduction.

Notesfiles support computer managed discussion forums. Discussions can have many different purposes and scopes: the notesfile system has been designed to be flexible enough to handle differing requirements.

Each notesfile discusses a single topic. The depth of discussion within a notesfile is ideally held constant. While some users may require a general discussion of personal workstations, a different group may desire detailed discussions about the I/O bus structure of the WICAT 68000 (a particular workstation). These discussions might well be separated into two different notesfiles.

Each notesfile contains a list of logically independent notes (called base notes). A note is a block of text with a comment or question intended to be seen by members of the notesfile community. The note display shows the text, its creation time, its title, the notesfile's title, the author's name (some notesfiles allow anonymous notes), the number of "responses", and optionally a "director message". Each base note can have a number of "responses": replies, retorts, further comments, criticism, or related questions concerning the base note. Thus, a notesfile contains an ordered list of ordered lists. This arrangement has historically been more convenient than other proposals (e.g., trees were studied on the PLATO (trademark of Control Data Corporation) system).

The concept of a notesfile was originally implemented at the University of Illinois, Urbana-Champaign, on the PLATO system. The UNIX (trademark of Bell Laboratories) notesfile system includes these ideas with adaptations and enhancements made possible by the UNIX environment.

The UNIX notesfile system was designed and implemented by Ray Essick at the University of Illinois, Urbana-Champaign. It provides users with the abilities to read notes and responses, write notes and responses, forward note text to other users (via mail) or other notesfiles, save note text in their own files, and sequence through a set of notesfiles seeing just new text. Each notesfile has a set of "directors" who manage the notesfile: they delete old notes, compress the file when needed, grant and restrict access to the notesfile, and set different notesfile parameters (e.g., title, "director message", policy note, whether notes' authors can be anonymous). Some notesfiles contain correspondence from other computers. Like the UNIX "USENET", notes and responses are exchanged (often over phone lines) with remote machines. The notesfile system provides automatic exchange and updating of notes in an arbitrarily connected network.

This document details the use of notesfiles from invocation through intersystem notes exchanges. The last chapter summarizes the entire set of commands for easy reference. An appendix contains detailed checklists for the installation of a notesfile system.

2 Using Notesfiles.

The notesfile system is invoked with a single command line. Most notesfile commands require only a single character (like the vi editor). Those that require more than one character are terminated by a carriage return.

2.1 Invocation.

Invoke the notesfile system with:

```
notes [ -sxi ] [-a subsequencer] [-t termtype ] [ -f nfile ] [ topic1 ] [ topic2 ... ]
```

The topic list (e.g., topic1) specifies the notesfiles to read. Invoking the notes system with NO



2.3 The -f Option.

The “-f” option of the notesfile system specifies a file of notesfile names to read. The file consists of lines containing notesfile names:

```
nfgripes
net.unix-wizards
net.general
fa.telecom
```

The names start at the left margin; they are indented here for readability. Wildcard characters (“*”, “?”, “[”, and “]”) are acceptable in this context. Full names such as “/usr/spool/notes/general” are also accepted. Notesfiles can be eliminated through the “!” feature as described in section 2.2. The sequencer mode can be changed (see section 2.8) by inserting a line of the form:

```
-s
```

Again, this starts at the left margin. The “s” can be any of: “s”, “x”, “i”, or “n”. When a line of this form is read from the file, the sequencer mode is set to the corresponding mode: The normal “s”equencer, the e“x”tended sequencer, the “i”ndex sequencer, and “n”o sequencer.

To always enter nfgripes, micronotes, and bicycle while only entering the networked notesfiles “net.*” when new notes are present, one might use “notes -f myfile” with this “myfile”:

```
-x
nfgripes
micronotes
bicycle
-s
net.*
```

2.4 General.

Almost all notesfile commands consist of exactly one character (no carriage return). Only commands that are longer than one character require a terminating carriage return (currently, choosing a note to read is the only non-single character command).

The commands were chosen to be easy to remember. Upper case forms of commands usually function like their lower case counterparts but with some additional feature or power (i.e., “w” writes a response, “W” includes the current displayed text in the response).

Some commands are available almost everywhere in the notesfile system. These include those for help, exiting, forking a shell, and making a comment for the suggestion box.

2.4.1 Help.

Typing “?” anywhere will list the available options in an abbreviated format.

2.4.2 Exiting.

Type “q” (“quit”) to leave the current notesfile. Capital “Q” leaves the current notesfile and refrains from entering your last entry time into the sequencer table (see section “The Sequencer”). The notesfile system proceeds to the next topic in the invocation list. The “k” and “K” keys function exactly as “q” and “Q”.

- Search for keywords within notes' titles.
- Search for notes/responses by a specific author.
- Go to another notesfile.
- Consult the notesfile's archive.
- Read the policy note.
- Check on anonymous and networked status.
- Register a complaint/suggestion about notesfiles.
- Fork a shell.
- Exit the notes program.
- Invoke notesfile director options (if the user is a director).

2.5.1 Scrolling the Index Page.

Scroll the index page by:

- +, <return>, <space> forward one page
- * forward to the most recent page (* is multiple +'s)
- backward one page
- = backward all the way (= is multiple -'s)

2.5.2 Choosing Notes & Responses.

While on the index page, choose a note to read by typing its number followed by a carriage return. (This is the only command that requires a carriage return after it.) Usually the space bar is used to scan text. To skip to a particular note or response, use the features below.

While reading a note, ";" or "+" advances to the first response of the note. The next note is displayed if there are no responses. The number keys ("1", "2", ... , "9") advance that many responses. If there are fewer responses, the last response is displayed. The return key skips the responses and goes to the next note. Press "-" or backspace to see the previous page of the current note; if the page currently displayed is the first, the notesfile program displays the first page of the previous note.

While a response is on the screen, the ";" and "+" keys display the next response. As with reading a note, if there are no further responses these keys advance to the next note. The number keys ("1", ... , "9") will advance the appropriate number of responses. If there are fewer responses, the last response is displayed. The "-" or backspace keys display the previous page of the current response. If the current page is the first page of the response, these keys display the first page of the previous response. Enter "=" to see the base note of the current note string. Press the return key to proceed to the next note.

2.6 Notes & Responses.

2.6.1 Reading Notes.

After selecting a note from the index page (or entering the notesfile with your "sequencer" on), the note is displayed. A sample display is shown below:

2.6.2 Reading Responses.

Response displays are similar to those of main notes with the exception that "Response x of y" replaces the note's title. The first response to note 15 is shown below:

```
Note 15           Workstation Discussion
koehler          Response 1 of 2      11:53 pm Dec 11, 1981
```

Does anyone have any insight about the relative speeds of the Winchester disks available on these systems? The previous disk seems to have track to track response times commensurate with reasonably fast 8" floppies. I wonder if some of the manufacturers are using disks that will not meet reasonable specifications for response time for these kinds of applications.

On the other hand, with intelligent layout of file sectors, the I/O system could romp and stomp on often used files...

=====

The commands for manipulating the text of a long response are the same as those for looking at long notes. Typing space will move to the next page. Typing "-" or backspace will display the previous page, within the same limitations as for reading notes (only 50 pages are kept). Press "=" to go back to the first page of the text.

The options available while reading responses include:

- Display the next, previous, or first page of the response.
- Go to a different response (usually the next one).
- Go to the next unread note/response.
- Reread the base note.
- Reread the previous note.
- Return to the index page.
- Copy the response to another notesfile.
- Mail the response to someone.
- Save the response in your file space.
- Talk to the response's author.
- Write another response to the note.
- Search for keywords in note titles.
- Search for notes/responses by particular authors.
- Delete the response (if you are its author or a file director).
- Edit the response (if it is yours and there are no later responses).
- Fork a shell
- Go to another notesfile.
- Register a suggestion or complaint about the notesfile program.
- Exit the notesfile program.

2.6.3 Writing Notes & Responses.

Write new base notes by hitting "w" while reading the index page. The notesfile system will then invoke an editor ("ed" by default; use either of the shell variables NFED or EDITOR to change it). After the prompt, compose the text you wish to enter, then write the text to the disk and leave the editor. The system will prompt you for various options if they are available: anonymity, director message status, and the note's title.

To write a response to a note type "w" while that note or any of its responses is displayed. The same steps used to write a base note should then be followed.



2.6.9 Editing Note Titles.

While reading a base note, type "e" ("edit") to change the note's title (provided you are the author of the note or a notesfile director).

2.6.10 Editing Notes/Responses.

"E" allows editing of the text of a note or response. It is not permitted to edit an article if it has subsequent responses or if it has been sent to the network. If the "later responses" are deleted, it is possible to edit the original text.

2.7 Other Commands.

2.7.1 Returning to the Index Page.

Type "i" ("index") while reading notes or responses to return to the index page.

2.7.2 Searching Titles for Keywords.

While reading, you can search backwards for keywords appearing in note titles. Typing "x" ("x is the unknown title") prompts for the substring to be found. Searching begins at the current note (or from the last note shown on the index page) and proceeds towards note 1. The search is insensitive to upper/lowercase distinctions. Use upper case "X" to continue the search. The search can be aborted by hitting the RUBOUT (or DELETE) key.

2.7.3 Searching for Authors.

The "a" command searches backwards for notes or responses written by a specific author. Notesfiles prompts for the author's name. The "A" command continues the search backwards. The author name may be preceded by an optional 'system!'. Abort the search by hitting the RUBOUT (or DELETE) key.

The entire name need not be specified when searching for articles by a particular author. Author searching uses substring searching. Searching for the author "john" will yield articles written by a local user "john", a remote user "somewhere!johnston", and any articles from the "uiucjohnny" machine. Author searching is case sensitive.

2.7.4 Stacking Notesfiles.

Sometimes it is useful to be able to glance at another notesfile while reading notes. Using "n", the user can save (stack) his current place and peruse another notesfile.

When on the index page or while reading notes/responses, type "n" ("nest") to read another notesfile. Notesfiles prompts for the notesfile to read. If the notesfile exists, the place is marked in the old notesfile and the new one's index is displayed.

Type any of the standard keys to leave the nested notesfile. Both "q" and "Q" leave the nested notesfile and return to the previously stacked notesfile. Control-d ("signoff") causes the notesfile program to exit regardless of the depth of nesting.

Sequencing is turned off in the new notesfile regardless of its state in the old notesfile. The depth of the stack of notesfiles is limited only by the amount of memory available to the user.



disabled, the “last time” information is not modified. The “last time” information for a particular notesfile is updated as that notesfile is exited; using “Q” or control-D later will have no effect on the sequencer information for notesfiles already read.

The “o” and “O” commands allow the user to modify the variable used to determine whether notes and responses are “new”. The “o” command allows the user to set this variable to any date he wishes. Use the “O” command to set this variable to show only notes and responses written that day. The “last time” file kept for each user is never modified by the “o” and “O” commands.

When no more new notes or responses exist, both the “j” and “J” commands will take the user to the index page. To exit the notesfile, use the “q” command. Exiting with “q” will update the user’s “last entry” time. Exiting with capital “Q” will NOT modify the “last entry” time for that notesfile (neither will control-D).

The “l” and “L” command behave similarly to “j” and “J”. The difference is that while “j” and “J” take the user to the last index page when no more new notes or responses exist, the “l” and “L” commands will leave the notesfile as if a “q” had been typed. Thus when no more new notes exist, the “l” command is like typing “jq”.

2.8.2 Alternate Sequencers.

If several people share a login account, it is convenient for each to have a set of sequencing timestamps. This is accomplished through the use of the subsequencer option of notesfiles.

Specifying the -a option and a subsequencer name causes notes to use a different sequencing timestamp file. Many different subsequencer names can be used with each login account.

The main sequencer file for a given account is distinct from each of its subsequencer files. Each of the subsequencer files is normally distinct. If the subsequencer names are not unique in their first 6 characters, subsequencer files may collide.

2.8.3 Automatic Sequencing.

An alternate entry to the notes program allows the user to invoke notes with the sequencer enabled and a list of notesfiles to be scanned with a single, simple command. The “autoseq” command is invoked by typing

```
autoseq
```

and reads the environment variable “NFSEQ” to find the names of all notesfiles to be scanned. On some systems, the “autoseq” command may be known as “readnotes”, “autonotes” or some similar variant; substitute the appropriate name in the following paragraphs. The “NFSEQ” variable should be defined in .profile for Bourne shell users as follows:

```
NFSEQ="pbnotes,micronotes,helpnotes,works"
export NFSEQ
```

For users of the C shell, the following line should be added to the .login file:

```
setenv NFSEQ "pbnotes,micronotes,helpnotes,works"
```

With NFSEQ assigned this value, a call to autoseq will process the notesfiles “pbnotes”, “micronotes”, “helpnotes”, and “works” with the sequencer turned on.

The full naming conventions, pattern matching capabilities, and ‘!’ exclusion described in

what screen handling conventions to use. In most cases the value will be correctly initialized by the system at login time.

- “SHELL” specifies which shell the user is running. This will almost always be set by the operating system.

3 Other Notesfile Utilities.

The notesfile distribution includes utility programs to provide hard copy output, additional interfaces to user programs, and statistics. They are described below.

3.1 Hard Copy Output.

The program “nfpri” sends to standard output a nicely formatted listing of the notesfile in its command line. Its format is:

```
nfpri [-lnn] [-p] [-t] topic [ note# ] [ note#-note# ] [ ... ]
```

The “-l” option specifies an alternate page size (the default is 66). The optional note number list specifies that only certain notes of the notesfile are to be printed. The list can specify individual notes and ranges. The notes are printed in the order specified.

The -p option specifies that each notestring is to begin on a new page. The -t option signifies that only a table of contents is to be generated.

3.2 Piped Insertion of Notes.

The nfpri program enters text from the standard input into a notesfile:

```
nfpri topic [-t title] [-d ] [-a ]
```

The -t option allows specification of a title. The -d and -a options specify the director and anonymous flags respectively (if available). If no title is specified, one is manufactured from the first line of the note.

3.3 User Subroutines.

3.3.1 Nfcomment.

The nfcomment subroutine is callable from a user’s C program. It allows any user program to enter text into a notesfile:

```
nfcomment (nfname, text, title, dirflag, anonflag)
```

The parameters are:

```
char *nfname; /* name of notesfile */
char *text; /* null terminated text to be entered */
char *title; /* if non-null, title of note */
int dirflag; /* != 0 -> director flag on (if allowed) */
int anonflag; /* != 0 -> anonymous note (if allowed) */
```



```

rbenotes on uiucdcs at 6:24 pm May 7, 1982
      NOTES      RESPS      TOTALS
Local Reads      359        115        474
Local Written    53          55        108
Networked in     0           0           0
Networked out    0           0           0
Network Dropped  0           0           0
Network Transmissions: 0  Network Receptions: 0
Orphaned Responses Received: 0  Entries into notesfile: 109
Total time in notesfile: 66.57 minutes Average Time/entry: 0.61 minutes
Created at 10:04 pm May 5, 1982, Used on 3 days
    
```

A combined set of statistics is produced at the end of listings of more than one notesfile. The statistics are largely self explanatory.

3.5 Checking for New Notes.

The checknotes program checks the notesfiles specified by the NFSEQ environment variable to determine if there are new notes. The exit code is arranged to make the program useful in shell scripts: 0 (TRUE) is there are new notes, 1 (FALSE) otherwise.

Use the “-q” option to receive a message

There are new notes

if one or more of the notesfiles have notes/responses written since the user’s last entry time into that notesfile.

The “-n” option is similar to the “-q” option, with the exception that it yields output when there are no new notes. The output of checknotes with the “-n” option is:

There are no new notes

Use “-v” to print the name of each notesfile with new notes/responses. The “-s” option is suitable for use in conditional expressions in shell scripts; no output is generated by this option.



A Tutorial Introduction to the UNIX Text Editor

Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Almost all text input on the UNIX† operating system is done with the text-editor *ed*. This memorandum is a tutorial guide to help beginners get started with text editing.

Although it does not cover everything, it does discuss enough for most users' day-to-day needs. This includes printing, appending, changing, deleting, moving and inserting entire lines of text; reading and writing files; context searching and line addressing; the substitute command; the global commands; and the use of special characters for advanced editing.

Introduction

Ed is a "text editor", that is, an interactive program for creating and modifying "text", using directions provided by a user at a terminal. The text is often a document like this one, or a program or perhaps data for a program.

This introduction is meant to simplify learning *ed*. The recommended way to learn *ed* is to read this document, simultaneously using *ed* to follow the examples, then to read the description in section I of the *UNIX Programmer's Manual*, all the while experimenting with *ed*. (Solicitation of advice from experienced users is also useful.)

Do the exercises! They cover material not completely discussed in the actual text. An appendix summarizes the commands.

Disclaimer

This is an introduction and a tutorial. For this reason, no attempt is made to cover more than a part of the facilities that *ed* offers (although this fraction includes the most useful and frequently used parts). When you have mastered the Tutorial, try *Advanced Editing on UNIX*. Also, there is not enough space to explain basic UNIX procedures. We will assume that you know how to log on to UNIX, and that you have at least a vague understanding of what a file is. For more on that, read *UNIX for Beginners*.

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

You must also know what character to type as the end-of-line on your particular terminal. This character is the RETURN key on most terminals. Throughout, we will refer to this character, whatever it is, as RETURN.

Getting Started

We'll assume that you have logged in to your system and it has just printed the prompt character, usually either a \$ or a %. The easiest way to get *ed* is to type

`ed` (followed by a return)

You are now ready to go – *ed* is waiting for you to tell it what to do.

Creating Text – the Append command "a"

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

When *ed* is first started, it is rather like working with a blank piece of paper – there is no text or information present. This must be supplied by the person using *ed*; it is usually done by typing in the text, or by reading it into *ed* from a file. We will



Exercise 1:

Enter *ed* and create some text using

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

Write it out using *w*. Then leave *ed* with the *q* command, and print the file, to see that everything worked. (To print a file, say

```
pr filename
```

or

```
cat filename
```

in response to the prompt character. Try both.)

Reading text from a file – the Edit command “e”

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

```
e junk
```

would fetch the entire contents of the file *junk* into the buffer, and respond

```
68
```

which is the number of characters in *junk*. *If anything was already in the buffer, it is deleted first.*

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

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

This way, you can simply say *w* from time to time, and be secure in the knowledge that if you got the file name right at the beginning, you are writing into the proper file each time.

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

```
f
```

ed would reply

```
junk
```

Reading text from a file – the Read command “r”

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

```
r junk
```

will read the file *junk* into the buffer; it adds it to the end of whatever is already in the buffer. So if you do a read after an edit:

```
e junk
r junk
```

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

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

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

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

Exercise 2:

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

```
ed filename
```

is exactly equivalent to

```
ed
e filename
```

What does

```
f filename
```

do?

Printing the contents of the buffer – the Print command “p”

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

```
p
```

The way this is done is as follows. Specify the lines where you want printing to begin and where you want it to end, separated by a comma, and followed by the letter *p*. Thus to print the first two lines of the buffer, for example, (that is, lines 1 through 2)

typing

.=

Ed will respond by printing the value of dot.

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

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

Deleting lines: the "d" command

Suppose you want to get rid of the three extra lines in the buffer. This is done by the *delete* command

d

Except that **d** deletes lines instead of printing them, its action is similar to that of **p**. The lines to be deleted are specified for **d** exactly as they are for **p**:

starting line, ending line d

Thus the command

4,\$d

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

1,\$p

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

Exercise 4:

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

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

Or filename

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

```
Oa
... text ...
.
```

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

```
.
w
```

Modifying text: the Substitute command "s"

We are now ready to try one of the most important of all commands - the substitute command

s

This is the command that is used to change individual words or letters within a line or group of lines. It is what you use, for example, for correcting spelling mistakes and typing errors.

Suppose that by a typing error, line 1 says

Now is th time

- the *e* has been left off *the*. You can use **s** to fix this up as follows:

1s/th/the/

This says: "in line 1, substitute for the characters *th* the characters *the*." To verify that it works (*ed* will not print the result automatically) say

p

and get

Now is the time

which is what you wanted. Notice that dot must have been set to the line where the substitution took place, since the **p** command printed that line. Dot is always set this way with the **s** command.

The general way to use the substitute command is

starting-line, ending-line s/change this/to this/

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in *all* the lines between *starting-line* and *ending-line*. Only the first occurrence on each line is changed, however. If you want to change *every* occurrence, see Exercise 5. The rules for line numbers are the same as those for **p**, except that dot is set to the last line changed. (But there is a trap for the unwary: if no substitution took place, dot is *not* changed. This causes an error ? as a warning.)

Thus you can say



There were three parts to that last command: context search for the desired line, make the substitution, print the line.

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

Suppose the buffer contains the three familiar lines

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

Then the `ed` line numbers

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

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

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

or

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

or

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

The choice is dictated only by convenience. You could print all three lines by, for instance

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

or

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

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

```
1,$p
```

but not if there were several hundred.)

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

Exercise 6:

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

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

Try context searching using `?text?` instead of `/text/`. This scans lines in the buffer in reverse order rather than normal. This is sometimes useful if you go too far while looking for some string of characters – it's an easy way to back up.

(If you get funny results with any of the characters

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

read the section on "Special Characters".)

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

```
/string/
```

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

```
//
```

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

```
/string1/s/string2/
```

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

```
??
```

means "scan backwards for the same expression."

Change and Insert – "c" and "i"

This section discusses the `change` command

```
c
```

which is used to change or replace a group of one or more lines, and the `insert` command

```
i
```

which is used for inserting a group of one or more lines.

"Change", written as

```
c
```

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

```
+.1,$c
```

```
. . . type the lines of text you want here . . .
```



match the string following v:

v/ /d

deletes every line that does not contain a blank.

Special Characters

You may have noticed that things just don't work right when you used some characters like ., *, \$, and others in context searches and the substitute command. The reason is rather complex, although the cure is simple. Basically, *ed* treats these characters as special, with special meanings. For instance, *in a context search or the first string of the substitute command only*, . means "any character," not a period, so

/x.y/

means "a line with an x, *any character*, and a y," *not* just "a line with an x, a period, and a y." A complete list of the special characters that can cause trouble is the following:

^ . \$ [* \

Warning: The backslash character \ is special to *ed*. For safety's sake, avoid it where possible. If you have to use one of the special characters in a substitute command, you can turn off its magic meaning temporarily by preceding it with the backslash. Thus

s/\\.*/backslash dot star/

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

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

/string/

finds **string** only if it is at the beginning of a line: it will find

string

but not

the string...

The dollar-sign \$ is just the opposite of the circumflex; it means the end of a line:

/string\$/

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

/string\$/

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

/^.\$/

finds a line containing exactly one character.

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

/x.y/

matches any of

x+y
x-y
x y
x.y

This is useful in conjunction with *, which is a repetition character; a* is a shorthand for "any number of a's," so .* matches any number of anything. This is used like this:

s/.*/stuff/

which changes an entire line, or

s/.,//

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

[is used with] to form "character classes"; for example,

/[0123456789]/

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

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

Now is the time

and you wanted to put parentheses around it. You could just retype the line, but this is tedious. Or you could say

s/(/(
s/\$)/)

using your knowledge of ^ and \$. But the easiest way uses the &:

s/.*/(&)/

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

s/.*/&? &!!/

to produce

Now is the time? Now is the time!!

You don't have to match the whole line, of course: if the buffer contains



Advanced Editing on UNIX

Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

(Updated for 4.3BSD by Mark Seiden)

ABSTRACT

This paper is meant to help secretaries, typists and programmers to make effective use of the UNIX† facilities for preparing and editing text. It provides explanations and examples of

- special characters, line addressing and global commands in the editor `ed`;
- commands for “cut and paste” operations on files and parts of files, including the `mv`, `cp`, `cat` and `rm` commands, and the `r`, `w`, `m` and `t` commands of the editor;
- editing scripts and editor-based programs like `grep` and `sed`.

Although the treatment is aimed at non-programmers, new users with any background should find helpful hints on how to get their jobs done more easily.

1. INTRODUCTION

Although UNIX provides remarkably effective tools for text editing, that by itself is no guarantee that everyone will automatically make the most effective use of them. In particular, people who are not computer specialists — typists, secretaries, casual users — often use the system less effectively than they might. (There is a good argument that new users would better use their time learning a display editor, like `vi`, or perhaps a version of `emacs`, like `jove`, rather than an editor as ignorant of display terminals as `ed`.)

This document is intended as a sequel to *A Tutorial Introduction to the UNIX Text Editor* [1], providing explanations and examples of how to edit using `ed` with less effort. (You should also be familiar with the material in *UNIX For Beginners* [2].) Further information on all commands discussed here can be found in section 1 of the *The UNIX User's Manual* [3].

Examples are based on observations of users and the difficulties they encounter. Topics covered include special characters in searches and substitute

commands, line addressing, the global commands, and line moving and copying. There are also brief discussions of effective use of related tools, like those for file manipulation, and those based on `ed`, like `grep` and `sed`.

A word of caution. There is only one way to learn to use something, and that is to *use* it. Reading a description is no substitute for trying something. A paper like this one should give you ideas about what to try, but until you actually try something, you will not learn it.

2. SPECIAL CHARACTERS

The editor `ed` is the primary interface to the system for many people, so it is worthwhile to know how to get the most out of `ed` for the least effort.

The next few sections will discuss shortcuts and labor-saving devices. Not all of these will be instantly useful to any one person, of course, but a few will be, and the others should give you ideas to store away for future use. And as always, until you try these things, they will remain theoretical knowledge, not something you have confidence in.

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



```
x+y
x-y
x□y
x.y
```

and so on. (We will use □ to stand for a space whenever we need to make it visible.)

Since '.' matches a single character, that gives you a way to deal with funny characters printed by I. Suppose you have a line that, when printed with the I command, appears as

```
.... th\07is ....
```

and you want to get rid of the \07 (which represents the bell character, by the way).

The most obvious solution is to try

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

but this will fail. (Try it.) The brute force solution, which most people would now take, is to re-type the entire line. This is guaranteed, and is actually quite a reasonable tactic if the line in question isn't too big, but for a very long line, re-typing is a bore. This is where the metacharacter '.' comes in handy. Since '\07' really represents a single character, if we say

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

the job is done. The '.' matches the mysterious character between the 'h' and the 'i', *whatever it is*.

Bear in mind that since '.' matches any single character, the command

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

converts the first character on a line into a '.', which very often is not what you intended.

As is true of many characters in ed, the '.' has several meanings, depending on its context. This line shows all three:

```
./././
```

The first '.' is a line number, the number of the line we are editing, which is called 'line dot'. (We will discuss line dot more in Section 3.) The second '.' is a metacharacter that matches any single character on that line. The third '.' is the only one that really is an honest literal period. On the *right* side of a substitution, '.' is not special. If you apply this command to the line

```
Now is the time.
```

the result will be

```
.ow is the time.
```

which is probably not what you intended.

The Backslash '\'

Since a period means 'any character', the question naturally arises of what to do when you really want a period. For example, how do you convert the line

```
Now is the time.
```

into

```
Now is the time?
```

The backslash '\' does the job. A backslash turns off any special meaning that the next character might have; in particular, '\' converts the '.' from a 'match anything' into a period, so you can use it to replace the period in

```
Now is the time.
```

like this:

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

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

The backslash can also be used when searching for lines that contain a special character. Suppose you are looking for a line that contains

```
.PP
```

The search

```
/./PP/
```

isn't adequate, for it will find a line like

```
THE APPLICATION OF ...
```

because the '.' matches the letter 'A'. But if you say

```
\/./PP/
```

you will find only lines that contain '.PP'.

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

```
\/\
```

won't work, because the '\' isn't a literal '\', but instead means that the second '/' no longer delimits the search. But by preceding a backslash with another one, you can search for a literal backslash. Thus

```
\/\
```

does work. Similarly, you can search for a forward slash '/' with

```
\/
```

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

The Star '*'

Suppose you have a line that looks like this:

text x y text

where *text* stands for lots of text, and there are some indeterminate number of spaces between the *x* and the *y*. Suppose the job is to replace all the spaces between *x* and *y* by a single space. The line is too long to retype, and there are too many spaces to count. What now?

This is where the metacharacter '*' comes in handy. A character followed by a star stands for as many consecutive occurrences of that character as possible. To refer to all the spaces at once, say

`s/x□*y/x□y/`

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

The star can be used with any character, not just space. If the original example was instead

text x-----y text

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

`s/x-*y/x□y/`

Finally, suppose that the line was

text x.....y text

Can you see what trap lies in wait for the unwary? If you blindly type

`s/x.*y/x□y/`

what will happen? The answer, naturally, is that it depends. If there are no other x's or y's on the line, then everything works, but it's blind luck, not good management. Remember that '.' matches *any* single character? Then '*.' matches as many single characters as possible, and unless you're careful, it can eat up a lot more of the line than you expected. If the line was, for example, like this:

text x text x.....y text y text

then saying

`s/x.*y/x□y/`

will take everything from the *first* 'x' to the *last* 'y', which, in this example, is undoubtedly more than you wanted.

The solution, of course, is to turn off the special meaning of '.' with '\.':

`s/x\.*y/x□y/`

Now everything works, for '\.*' means 'as many periods as possible'.

There are times when the pattern '*.' is exactly what you want. For example, to change

Now is the time for all good men

into

Now is the time.

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

`s/□for.*./`

There are a couple of additional pitfalls associated with '*' that you should be aware of. Most notable is the fact that 'as many as possible' means *zero* or more. The fact that zero is a legitimate possibility is sometimes rather surprising. For example, if our line contained

text xy text x y text

and we said

`s/x□*y/x□y/`

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

The way around this, if it matters, is to specify a pattern like

`/x□□*y/`

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

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

`s/x*/y/g`

when applied to the line

abcdef

produces

yaybycydeyfy

which is almost certainly not what was intended. The reason for this behavior is that zero is a legal number of matches, and there are no x's at the beginning of the line (so that gets converted into a 'y'), nor between the 'a' and the 'b' (so that gets converted into a 'y'), nor ... and so on. Make sure you really want zero matches; if not, in this case write

`s/xx*/y/g`

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



you can break it between the 'x' and the 'y' like this:

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

This is actually a single command, although it is typed on two lines. Bearing in mind that '\' turns off special meanings, it seems relatively intuitive that a '\' at the end of a line would make the new-line there no longer special.

You can in fact make a single line into several lines with this same mechanism. As a large example, consider underlining the word 'very' in a long line by splitting 'very' onto a separate line, and preceding it by the **roff** or **nroff** formatting command '.ul'.

```
text a very big text
```

The command

```
s/□very□/  
.ul\  
very\  
/
```

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

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

Joining Lines

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

```
Now is  
□the time
```

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

```
j
```

joins them together. No blanks are added, which is why we carefully showed a blank at the beginning of the second line.

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

```
1,$jp
```

joins all the lines into one big one and prints it. (More on line numbers in Section 3.)

Rearranging a Line with \ (... \)

(This section should be skipped on first reading.) Recall that '&' is a shorthand that stands for whatever was matched by the left side of an **s** command. In much the same way you can capture

separate pieces of what was matched; the only difference is that you have to specify on the left side just what pieces you're interested in.

Suppose, for instance, that you have a file of lines that consist of names in the form

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

and so on, and you want the initials to precede the name, as in

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

It is possible to do this with a series of editing commands, but it is tedious and error-prone. (It is instructive to figure out how it is done, though.)

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

The command

```
1,$s/\(("[,]*\),□*(.*)\)/2□1/
```

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

Of course, with any editing sequence this complicated, it's foolhardy to simply run it and hope. The global commands **g** and **v** discussed in section 4 provide a way for you to print exactly those lines which were affected by the substitute command, and thus verify that it did what you wanted in all cases.

3. LINE ADDRESSING IN THE EDITOR

The next general area we will discuss is that of line addressing in **ed**, that is, how you specify what lines are to be affected by editing commands. We have already used constructions like

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

to specify a change on all lines. And most users are long since familiar with using a single newline (or return) to print the next line, and with

```
/thing/
```

to find a line that contains 'thing'. Less familiar, surprisingly enough, is the use of

```
?thing?
```

to scan *backwards* for the previous occurrence of

What happens if there was no 'thing'? Then you are left right where you were — dot is unchanged. This is also true if you were sitting on the only 'thing' when you issued the command. The same rules hold for searches that use '?...?'; the only difference is the direction in which you search.

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

The line-changing commands **a**, **c** and **i** by default all affect the current line — if you give no line number with them, **a** appends text after the current line, **c** changes the current line, and **i** inserts text before the current line.

a, **c**, and **i** behave identically in one respect — when you stop appending, changing or inserting, dot points at the last line entered. This is exactly what you want for typing and editing on the fly. For example, you can say

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

without specifying any line number for the substitute command or for the second append command. Or you can say

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

You should experiment to determine what happens if you add *no* lines with **a**, **c** or **i**.

The **r** command will read a file into the text being edited, either at the end if you give no address, or after the specified line if you do. In either case, dot points at the last line read in. Remember that you can even say **0r** to read a file in at the beginning of the text. (You can also say **0a** or **1i** to start adding text at the beginning.)

The **w** command writes out the entire file. If you precede the command by one line number, that line is written, while if you precede it by two line numbers, that range of lines is written. The **w** command does *not* change dot: the current line remains the same, regardless of what lines are written. This is true even if you say something like

```
1/^\.AB/,/^\.AE/w abstract
```

which involves a context search.

Since the **w** command is so easy to use, you should save what you are editing regularly as you go along just in case the system crashes, or in case you do something foolish, like clobbering what you're editing.

The least intuitive behavior, in a sense, is that of the **s** command. The rule is simple — you are left sitting on the last line that got changed. If there were no changes, then dot is unchanged.

To illustrate, suppose that there are three lines in the buffer, and you are sitting on the middle one:

```
x1
x2
x3
```

Then the command

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

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

```
x1
y2
y3
```

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

Semicolon ';' :

Searches with '/.../' and '?...?' start at the current line and move forward or backward respectively until they either find the pattern or get back to the current line. Sometimes this is not what is wanted. Suppose, for example, that the buffer contains lines like this:

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

Starting at line 1, one would expect that the command

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

prints all the lines from the 'ab' to the 'bc' inclusive. Actually this is not what happens. *Both* searches (for 'a' and for 'b') start from the same point, and thus they both find the line that contains 'ab'. The result is to print a single line. Worse, if there had been a line with a 'b' in it before the 'ab' line, then the print command would be in error, since the



took place.

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

```
g/\,PP/+
```

prints the line that follows each `.PP` command (the signal for a new paragraph in some formatting packages). Remember that `+` means 'one line past dot'. And

```
g/topic/?\,SH?1
```

searches for each line that contains 'topic', scans backwards until it finds a line that begins `.SH` (a section heading) and prints the line that follows that, thus showing the section headings under which 'topic' is mentioned. Finally,

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

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

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

Multi-line Global Commands

It is possible to do more than one command under the control of a global command, although the syntax for expressing the operation is not especially natural or pleasant. As an example, suppose the task is to change 'x' to 'y' and 'a' to 'b' on all lines that contain 'thing'. Then

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

is sufficient. The `\` signals the `g` command that the set of commands continues on the next line; it terminates on the first line that does not end with `\`. (As a minor blemish, you can't use a substitute command to insert a newline within a `g` command.)

You should watch out for this problem: the command

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

does *not* work as you expect. The remembered pattern is the last pattern that was actually executed, so sometimes it will be 'x' (as expected), and sometimes it will be 'a' (not expected). You must spell it out, like this:

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

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

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

There is no need for a final line containing a `.` to terminate the `i` command, unless there are further commands being done under the global. On the other hand, it does no harm to put it in either.

5. CUT AND PASTE WITH UNIX COMMANDS

One editing area in which non-programmers seem not very confident is in what might be called 'cut and paste' operations — changing the name of a file, making a copy of a file somewhere else, moving a few lines from one place to another in a file, inserting one file in the middle of another, splitting a file into pieces, and splicing two or more files together.

Yet most of these operations are actually quite easy, if you keep your wits about you and go cautiously. The next several sections talk about cut and paste. We will begin with the UNIX commands for moving entire files around, then discuss `ed` commands for operating on pieces of files.

Changing the Name of a File

You have a file named 'memo' and you want it to be called 'paper' instead. How is it done?

The UNIX program that renames files is called `mv` (for 'move'); it 'moves' the file from one name to another, like this:

```
mv memo paper
```

That's all there is to it: `mv` from the old name to the new name.

```
mv oldname newname
```

Warning: if there is already a file around with the new name, its present contents will be silently clobbered by the information from the other file. The one exception is that you can't move a file to itself

```
mv x x
```

is illegal.

Making a Copy of a File

Sometimes what you want is a copy of a file — an entirely fresh version. This might be because you want to work on a file, and yet save a copy in case something gets fouled up, or just because you're

```
cat good1 >>good
```

and 'good1' is added to the end of 'good'. (And if 'good' didn't exist, this makes a copy of 'good1' called 'good'.)

6. CUT AND PASTE WITH THE EDITOR

Now we move on to manipulating pieces of files — individual lines or groups of lines. This is another area where new users seem unsure of themselves.

Filenames

The first step is to ensure that you know the `ed` commands for reading and writing files. Of course you can't go very far without knowing `r` and `w`. Equally useful, but less well known, is the 'edit' command `e`. Within `ed`, the command

```
e newfile
```

says 'I want to edit a new file called *newfile*, without leaving the editor.' The `e` command discards whatever you're currently working on and starts over on *newfile*. It's exactly the same as if you had quit with the `q` command, then re-entered `ed` with a new file name, except that if you have a pattern remembered, then a command like `//` will still work.

If you enter `ed` with the command

```
ed file
```

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

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

(and so on) does a series of edits on various files without ever leaving `ed` and without typing the name of any file more than once. (As an aside, if you examine the sequence of commands here, you can see why many UNIX systems use `e` as a synonym for `ed`.)

You can find out the remembered file name at any time with the `f` command; just type `f` without a file name. You can also change the name of the remembered file name with `f`; a useful sequence is

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

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

Inserting One File into Another

Suppose you have a file called 'memo', and you want the file called 'table' to be inserted just after the reference to Table 1. That is, in 'memo' somewhere is a line that says

```
Table 1 shows that ...
```

and the data contained in 'table' has to go there, probably so it will be formatted properly by `nroff` or `troff`. Now what?

This one is easy. Edit 'memo', find 'Table 1', and add the file 'table' right there:

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

The critical line is the last one. As we said earlier, the `r` command reads a file; here you asked for it to be read in right after line dot. An `r` command without any address adds lines at the end, so it is the same as `$r`.

Writing out Part of a File

The other side of the coin is writing out part of the document you're editing. For example, maybe you want to copy out into a separate file that table from the previous example, so it can be formatted and tested separately. Suppose that in the file being edited we have

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

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

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

and the job is done. If you are confident, you can do it all at once with

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

and now you have two copies, one in the file you're still editing, one in the file 'table' you've just written.

The point is that the `w` command can write out a group of lines, instead of the whole file. In fact, you can write out a single line if you like; just give one line number instead of two. For example, if you have just typed a horribly complicated line and you know that it (or something like it) is going to be needed later, then save it — don't re-type it. In the editor, say

Copying Lines

We mentioned earlier the idea of saving a line that was hard to type or used often, so as to cut down on typing time. Of course this could be more than one line; then the saving is presumably even greater.

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

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

```
1,$t$
```

duplicates the entire contents that you are editing. A more common use for `t` is for creating a series of lines that differ only slightly. For example, you can say

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

and so on.

The Temporary Escape ‘!’

Sometimes it is convenient to be able to temporarily escape from the editor to do some other UNIX command, perhaps one of the file copy or move commands discussed in section 5, without leaving the editor. The 'escape' command `!` provides a way to do this.

If you say

```
!any UNIX command
```

your current editing state is suspended, and the UNIX command you asked for is executed. When the command finishes, `ed` will signal you by printing another `!`; at that point you can resume editing.

You can really do *any* UNIX command, including another `ed`. (This is quite common, in fact.) In this case, you can even do another `!`.

On Berkeley UNIX systems, there is an additional (and preferable) mechanism called *job control* which lets you suspend your edit session (or, for that matter, any program), return to the shell from which you invoked that program, and issue any commands, then resume the program from the point where it was stopped. See *An Introduction to the C Shell* for more details.

7. SUPPORTING TOOLS

There are several tools and techniques that go along with the editor, all of which are relatively easy once you know how `ed` works, because they are all based on the editor. In this section we will give some fairly cursory examples of these tools, more to indicate their existence than to provide a complete tutorial. More information on each can be found in [3].

Grep

Sometimes you want to find all occurrences of some word or pattern in a set of files, to edit them or perhaps just to verify their presence or absence. It may be possible to edit each file separately and look for the pattern of interest, but if there are many files this can get very tedious, and if the files are really big, it may be impossible because of limits in `ed`.

The program `grep` was invented to get around these limitations. The search patterns that we have described in the paper are often called 'regular expressions', and 'grep' stands for

```
g/re/p
```

That describes exactly what `grep` does — it prints every line in a set of files that contains a particular pattern. Thus

```
grep 'thing' file1 file2 file3 ...
```

finds 'thing' wherever it occurs in any of the files 'file1', 'file2', etc. `grep` also indicates the file in which the line was found, so you can later edit it if you like.

The pattern represented by 'thing' can be any pattern you can use in the editor, since `grep` and `ed` use exactly the same mechanism for pattern searching. It is wisest always to enclose the pattern in the single quotes '...' if it contains any non-alphabetic characters, since many such characters also mean something special to the UNIX command interpreter (the 'shell'). If you don't quote them, the command interpreter will try to interpret them before `grep` gets a chance.

There is also a way to find lines that *don't* contain a pattern:

```
grep -v 'thing' file1 file2 ...
```

finds all lines that don't contain 'thing'. The `-v` must occur in the position shown. Given `grep` and `grep -v`, it is possible to do things like selecting all lines that contain some combination of patterns. For example, to get all lines that contain 'x' but not 'y':

```
grep x file... | grep -v y
```

(The notation `|` is a 'pipe', which causes the output

Edit: A Tutorial

Ricki Blau

James Joyce

Computing Services
University of California
Berkeley, California 94720

ABSTRACT

This narrative introduction to the use of the text editor *edit* assumes no prior familiarity with computers or with text editing. Its aim is to lead the beginning UNIX† user through the fundamental steps of writing and revising a file of text. Edit, a version of the text editor *ex*, was designed to provide an informative environment for new and casual users.

We welcome comments and suggestions about this tutorial and the UNIX documentation in general.

September 1981

U
S 14
D

†UNIX is a trademark of Bell Laboratories.

Introduction

Text editing using a terminal connected to a computer allows you to create, modify, and print text easily. A *text editor* is a program that assists you as you create and modify text. The text editor you will learn here is named *edit*. Creating text using *edit* is as easy as typing it on an electric typewriter. Modifying text involves telling the text editor what you want to add, change, or delete. You can review your text by typing a command to print the file contents as they are currently. Another program (which we do not discuss in this document), a text formatter, rearranges your text for you into "finished form."

These lessons assume no prior familiarity with computers or with text editing. They consist of a series of text editing sessions which lead you through the fundamental steps of creating and revising text. After scanning each lesson and before beginning the next, you should try the examples at a terminal to get a feeling for the actual process of text editing. If you set aside some time for experimentation, you will soon become familiar with using the computer to write and modify text. In addition to the actual use of the text editor, other features of UNIX will be very important to your work. You can begin to learn about these other features by reading one of the other tutorials that provide a general introduction to the system. You will be ready to proceed with this lesson as soon as you are familiar with (1) your terminal and its special keys, (2) how to login, (3) and the ways of correcting typing errors. Let's first define some terms:

- program A set of instructions, given to the computer, describing the sequence of steps the computer performs in order to accomplish a specific task. The task must be specific, such as balancing your checkbook or editing your text. A general task, such as working for world peace, is something we can all do, but not something we can currently write programs to do.
- UNIX UNIX is a special type of program, called an operating system, that supervises the machinery and all other programs comprising the total computer system.
- edit *edit* is the name of the UNIX text editor you will be learning to use, and is a program that aids you in writing or revising text. *Edit* was designed for beginning users, and is a simplified version of an editor named *ex*.
- file Each UNIX account is allotted space for the permanent storage of information, such as programs, data or text. A file is a logical unit of data, for example, an essay, a program, or a chapter from a book, which is stored on a computer system. Once you create a file, it is kept until you instruct the system to remove it. You may create a file during one UNIX session, end the session, and return to use it at a later time. Files contain anything you choose to write and store in them. The sizes of files vary to suit your needs; one file might hold only a single number, yet another might contain a very long document or program. The only way to save information from one session to the next is to store it in a file, which you will learn in Session 1.
- filename Filenames are used to distinguish one file from another, serving the same purpose as the labels of manila folders in a file cabinet. In order to write or access information in a file, you use the name of that file in a UNIX command, and the system will automatically locate the file.
- disk Files are stored on an input/output device called a disk, which looks something like a stack of phonograph records. Each surface is coated with a material similar to that on magnetic recording tape, and information is recorded on it.
- buffer A temporary work space, made available to the user for the duration of a session of text editing and used for creating and modifying the text file. We can think of the buffer as a blackboard that is erased after each class, where each session with the editor is a class.

```
% edit text (followed by a RETURN)
"text" No such file or directory
:
```

If you typed the command correctly, you will now be in communication with edit. Edit has set aside a buffer for use as a temporary working space during your current editing session. Since "text" is a new file we are about to create the editor was unable to find that file, which it confirms by saying:

```
"text" No such file or directory
```

On the next line appears edit's prompt ":", announcing that you are in *command mode* and edit expects a command from you. You may now begin to create the new file.

The "Command not found" message

If you misspelled edit by typing, say, "editor", this might appear:

```
% editor
editor: Command not found
%
```

Your mistake in calling edit "editor" was treated by UNIX as a request for a program named "editor". Since there is no program named "editor", UNIX reported that the program was "not found". A new % indicates that UNIX is ready for another command, and you may then enter the correct command.

A summary

Your exchange with UNIX as you logged in and made contact with edit should look something like this:

```
login: susan
Password:
... A Message of General Interest ...
% edit text
"text" No such file or directory
:
```

Entering text

You may now begin entering text into the buffer. This is done by *appending* (or adding) text to whatever is currently in the buffer. Since there is nothing in the buffer at the moment, you are appending text to nothing; in effect, since you are adding text to nothing you are creating text. Most edit commands have two equivalent forms: a word that suggests what the command does, and a shorter abbreviation of that word. Many beginners find the full command names easier to remember at first, but once you are familiar with editing you may prefer to type the shorter abbreviations. The command to input text is "append". (It may be abbreviated "a".) Type **append** and press the RETURN key.

```
% edit text
:append
```

Messages from edit

If you make a mistake in entering a command and type something that edit does not recognize, edit will respond with a message intended to help you diagnose your error. For example, if you misspell the command to input text by typing, perhaps, "add" instead of "append" or "a", you will receive this message:



on. You may immediately begin to retype the line. This, unfortunately, does not work after you type the line and press RETURN. To make corrections in lines that have been completed, it is necessary to use the editing commands covered in the next sessions.

Writing text to disk

You are now ready to edit the text. One common operation is to write the text to disk as a file for safekeeping after the session is over. This is the only way to save information from one session to the next, since the editor's buffer is temporary and will last only until the end of the editing session. Learning how to write a file to disk is second in importance only to entering the text. To write the contents of the buffer to a disk file, use the command "write" (or its abbreviation "w"):

```
: write
```

Edit will copy the contents of the buffer to a disk file. If the file does not yet exist, a new file will be created automatically and the presence of a "[New file]" will be noted. The newly-created file will be given the name specified when you entered the editor, in this case "text". To confirm that the disk file has been successfully written, edit will repeat the filename and give the number of lines and the total number of characters in the file. The buffer remains unchanged by the "write" command. All of the lines that were written to disk will still be in the buffer, should you want to modify or add to them.

Edit must have a name for the file to be written. If you forgot to indicate the name of the file when you began to edit, edit will print in response to your write command:

```
No current filename
```

If this happens, you can specify the filename in a new write command:

```
: write text
```

After the "write" (or "w"), type a space and then the name of the file.

Signing off

We have done enough for this first lesson on using the UNIX text editor, and are ready to quit the session with edit. To do this we type "quit" (or "q") and press RETURN:

```
: write
"text" [New file] 3 lines, 90 characters
: quit
%
```

The % is from UNIX to tell you that your session with edit is over and you may command UNIX further. Since we want to end the entire session at the terminal, we also need to exit from UNIX. In response to the UNIX prompt of "%" type the command

```
% logout
```

This will end your session with UNIX, and will ready the terminal for the next user. It is always important to type **logout** at the end of a session to make absolutely sure no one could accidentally stumble into your abandoned session and thus gain access to your files, tempting even the most honest of souls.

This is the end of the first session on UNIX text editing.

Listing what's in the buffer (p)

Having appended text to what you wrote in Session 1, you might want to see all the lines in the buffer. To print the contents of the buffer, type the command:

```
:1,$p
```

The “1”† stands for line 1 of the buffer, the “\$” is a special symbol designating the last line of the buffer, and “p” (or **print**) is the command to print from line 1 to the end of the buffer. The command “1,\$p” gives you:

```
This is some sample text.
And thiss is some more text.
Text editing is strange, but nice.
This is text added in Session 2.
It doesn't mean much here, but
it does illustrate the editor.
```

Occasionally, you may accidentally type a character that can't be printed, which can be done by striking a key while the CTRL key is pressed. In printing lines, edit uses a special notation to show the existence of non-printing characters. Suppose you had introduced the non-printing character “control-A” into the word “illustrate” by accidentally pressing the CTRL key while typing “a”. This can happen on many terminals because the CTRL key and the “A” key are beside each other. If your finger presses between the two keys, control-A results. When asked to print the contents of the buffer, edit would display

```
it does illustr^Ate the editor.
```

To represent the control-A, edit shows “^A”. The sequence “^” followed by a capital letter stands for the one character entered by holding down the CTRL key and typing the letter which appears after the “^”. We'll soon discuss the commands that can be used to correct this typing error.

In looking over the text we see that “this” is typed as “thiss” in the second line, a deliberate error so we can learn to make corrections. Let's correct the spelling.

Finding things in the buffer

In order to change something in the buffer we first need to find it. We can find “thiss” in the text we have entered by looking at a listing of the lines. Physically speaking, we search the lines of text looking for “thiss” and stop searching when we have found it. The way to tell edit to search for something is to type it inside slash marks:

```
:/thiss/
```

By typing `/thiss/` and pressing RETURN, you instruct edit to search for “thiss”. If you ask edit to look for a pattern of characters which it cannot find in the buffer, it will respond “Pattern not found”. When edit finds the characters “thiss”, it will print the line of text for your inspection:

```
And thiss is some more text.
```

Edit is now positioned in the buffer at the line it just printed, ready to make a change in the line.

†The numeral “one” is the top left-most key, and should not be confused with the letter “el”.



Substitute pattern match failed

indicating that your instructions could not be carried out. When edit does find the characters that you want to change, it will make the substitution and automatically print the changed line, so that you can check that the correct substitution was made. In the example,

```
:2s/thiss/this/
```

And this is some more text.

line 2 (and line 2 only) will be searched for the characters "thiss", and when the first exact match is found, "thiss" will be changed to "this". Strictly speaking, it was not necessary above to specify the number of the line to be changed. In

```
:s/thiss/this/
```

edit will assume that we mean to change the line where we are currently located ("."). In this case, the command without a line number would have produced the same result because we were already located at the line we wished to change.

For another illustration of the substitute command, let us choose the line:

Text editing is strange, but nice.

You can make this line a bit more positive by taking out the characters "strange, but " so the line reads:

Text editing is nice.

A command that will first position edit at the desired line and then make the substitution is:

```
:/strange/s/strange, but //
```

What we have done here is combine our search with our substitution. Such combinations are perfectly legal, and speed up editing quite a bit once you get used to them. That is, you do not necessarily have to use line numbers to identify a line to edit. Instead, you may identify the line you want to change by asking edit to search for a specified pattern of letters that occurs in that line. The parts of the above command are:

<code>/strange/</code>	tells edit to find the characters "strange" in the text
<code>s</code>	tells edit to make a substitution
<code>/strange, but //</code>	substitutes nothing at all for the characters "strange, but "

You should note the space after "but" in `/strange, but /`. If you do not indicate that the space is to be taken out, your line will read:

Text editing is nice.

which looks a little funny because of the extra space between "is" and "nice". Again, we realize from this that a blank space is a real character to a computer, and in editing text we need to be aware of spaces within a line just as we would be aware of an "a" or a "4".

Another way to list what's in the buffer (z)

Although the print command is useful for looking at specific lines in the buffer, other commands may be more convenient for viewing large sections of text. You can ask to see a screen full of text at a time by using the command `z`. If you type

```
:1z
```

edit will start with line 1 and continue printing lines, stopping either when the screen of your terminal is full or when the last line in the buffer has been printed. If you want to read the next segment of text, type the command



Session 3

Bringing text into the buffer (e)

Login to UNIX and make contact with edit. You should try to login without looking at the notes, but if you must then by all means do.

Did you remember to give the name of the file you wanted to edit? That is, did you type

```
% edit text
```

or simply

```
% edit
```

Both ways get you in contact with edit, but the first way will bring a copy of the file named "text" into the buffer. If you did forget to tell edit the name of your file, you can get it into the buffer by typing:

```
: e text
"text" 6 lines, 171 characters
```

The command **edit**, which may be abbreviated **e**, tells edit that you want to erase anything that might already be in the buffer and bring a copy of the file "text" into the buffer for editing. You may also use the edit (**e**) command to change files in the middle of an editing session, or to give edit the name of a new file that you want to create. Because the edit command clears the buffer, you will receive a warning if you try to edit a new file without having saved a copy of the old file. This gives you a chance to write the contents of the buffer to disk before editing the next file.

Moving text in the buffer (m)

Edit allows you to move lines of text from one location in the buffer to another by means of the **move (m)** command. The first two examples are for illustration only, though after you have read this Session you are welcome to return to them for practice. The command

```
:2,4m$
```

directs edit to move lines 2, 3, and 4 to the end of the buffer (\$). The format for the move command is that you specify the first line to be moved, the last line to be moved, the move command "m", and the line after which the moved text is to be placed. So,

```
:1,3m6
```

would instruct edit to move lines 1 through 3 (inclusive) to a location after line 6 in the buffer. To move only one line, say, line 4, to a location in the buffer after line 5, the command would be "4m5".

Let's move some text using the command:

```
:5,$m1
2 lines moved
it does illustrate the editor.
```

After executing a command that moves more than one line of the buffer, edit tells how many lines were affected by the move and prints the last moved line for your inspection. If you want to see more than just the last line, you can then use the print (**p**), **z**, or number (**nu**) command to view more text. The buffer should now contain:

```
This is some sample text.
It doesn't mean much here, but
it does illustrate the editor.
And this is some more text.
Text editing is nice.
This is text added in Session 2.
```

This is some sample text.
 And this is some more text.
 Text editing is nice.
 It doesn't mean much here, but
 it does illustrate the editor.
 And this is some more text.
 Text editing is nice.
 This is text added in Session 2.
 It doesn't mean much here, but

To delete both lines 2 and 3:

And this is some more text.
 Text editing is nice.

you type

```
:2,3d
2 lines deleted
```

which specifies the range of lines from 2 to 3, and the operation on those lines — “d” for delete. If you delete more than one line you will receive a message telling you the number of lines deleted, as indicated in the example above.

The previous example assumes that you know the line numbers for the lines to be deleted. If you do not you might combine the search command with the delete command:

```
:/And this is some/,/Text editing is nice./d
```

A word or two of caution

In using the search function to locate lines to be deleted you should be **absolutely sure** the characters you give as the basis for the search will take edit to the line you want deleted. Edit will search for the first occurrence of the characters starting from where you last edited — that is, from the line you see printed if you type dot (.).

A search based on too few characters may result in the wrong lines being deleted, which edit will do as easily as if you had meant it. For this reason, it is usually safer to specify the search and then delete in two separate steps, at least until you become familiar enough with using the editor that you understand how best to specify searches. For a beginner it is not a bad idea to double-check each command before pressing RETURN to send the command on its way.

Undo (u) to the rescue

The **undo (u)** command has the ability to reverse the effects of the last command that changed the buffer. To undo the previous command, type “u” or “undo”. Undo can rescue the contents of the buffer from many an unfortunate mistake. However, its powers are not unlimited, so it is still wise to be reasonably careful about the commands you give.

It is possible to undo only commands which have the power to change the buffer — for example, delete, append, move, copy, substitute, and even undo itself. The commands write (w) and edit (e), which interact with disk files, cannot be undone, nor can commands that do not change the buffer, such as print. Most importantly, the **only** command that can be reversed by undo is the last “undoable” command you typed. You can use control-H and @ to change commands while you are typing them, and undo to reverse the effect of the commands after you have typed them and pressed RETURN.

To illustrate, let's issue an undo command. Recall that the last buffer-changing command we gave deleted the lines formerly numbered 2 and 3. Typing undo at this moment will reverse the effects of the deletion, causing those two lines to be replaced in the buffer.

At end-of-file

or

Not that many lines in buffer

Similarly, if you try to move to a position before the first line, edit will print one of these messages:

Nonzero address required on this command

or

Negative address - first buffer line is 1

The number associated with a buffer line is the line's "address", in that it can be used to locate the line.

Changing lines (c)

You can also delete certain lines and insert new text in their place. This can be accomplished easily with the **change (c)** command. The change command instructs edit to delete specified lines and then switch to text input mode to accept the text that will replace them. Let's say you want to change the first two lines in the buffer:

This is some sample text.
And this is some more text.

to read

This text was created with the UNIX text editor.

To do so, you type:

```
:1,2c
2 lines changed
This text was created with the UNIX text editor.
:
:
```

In the command **1,2c** we specify that we want to change the range of lines beginning with 1 and ending with 2 by giving line numbers as with the print command. These lines will be deleted. After you type RETURN to end the change command, edit notifies you if more than one line will be changed and places you in text input mode. Any text typed on the following lines will be inserted into the position where lines were deleted by the change command. **You will remain in text input mode until you exit in the usual way, by typing a period alone on a line.** Note that the number of lines added to the buffer need not be the same as the number of lines deleted.

This is the end of the third session on text editing with UNIX.

```
:/text/s/text/texts/
```

as we have done in the past, or a somewhat abbreviated command:

```
:/text/s//texts/
```

In this example, the characters to be changed are not specified – there are no characters, not even a space, between the two slash marks that indicate what is to be changed. This lack of characters between the slashes is taken by the editor to mean “use the characters we last searched for as the characters to be changed.”

Similarly, the last context search may be repeated by typing a pair of slashes with nothing between them:

```
:/does/
It doesn't mean much here, but
://
it does illustrate the editor.
```

(You should note that the search command found the characters “does” in the word “doesn’t” in the first search request.) Because no characters are specified for the second search, the editor scans the buffer for the next occurrence of the characters “does”.

Edit normally searches forward through the buffer, wrapping around from the end of the buffer to the beginning, until the specified character string is found. If you want to search in the reverse direction, use question marks (?) instead of slashes to surround the characters you are searching for.

It is also possible to repeat the last substitution without having to retype the entire command. An ampersand (&) used as a command repeats the most recent substitute command, using the same search and replacement patterns. After altering the current line by typing

```
:s/text/texts/
```

you type

```
:/text/&
```

or simply

```
://&
```

to make the same change on the next line in the buffer containing the characters “text”.

Special characters

Two characters have special meanings when used in specifying searches: “\$” and “^”. “\$” is taken by the editor to mean “end of the line” and is used to identify strings that occur at the end of a line.

```
:g/text.$/s//material./p
```

tells the editor to search for all lines ending in “text.” (and nothing else, not even a blank space), to change each final “text.” to “material.”, and print the changed lines.

The symbol “^” indicates the beginning of a line. Thus,

```
:s/^1. /
```

instructs the editor to insert “1.” and a space at the beginning of the current line.

The characters “\$” and “^” have special meanings only in the context of searching. At other times, they are ordinary characters. If you ever need to search for a character that has a special meaning, you must indicate that the character is to lose temporarily its special significance by typing another special character, the backslash (\), before it.

```
:w
"text" 4 lines, 88 characters
:f
"text" line 3 of 4 --75%--
```

Reading additional files (r)

The **read (r)** command allows you to add the contents of a file to the buffer at a specified location, essentially copying new lines between two existing lines. To use it, specify the line after which the new text will be placed, the **read (r)** command, and then the name of the file. If you have a file named "example", the command

```
:$r example
"example" 18 lines, 473 characters
```

reads the file "example" and adds it to the buffer after the last line. The current filename is not changed by the read command.

Writing parts of the buffer

The **write (w)** command can write all or part of the buffer to a file you specify. We are already familiar with writing the entire contents of the buffer to a disk file. To write only part of the buffer onto a file, indicate the beginning and ending lines before the write command, for example

```
:45,$w ending
```

Here all lines from 45 through the end of the buffer are written onto the file named *ending*. The lines remain in the buffer as part of the document you are editing, and you may continue to edit the entire buffer. Your original file is unaffected by your command to write part of the buffer to another file. Edit still remembers whether you have saved changes to the buffer in your original file or not.

Recovering files

Although it does not happen very often, there are times UNIX stops working because of some malfunction. This situation is known as a *crash*. Under most circumstances, edit's crash recovery feature is able to save work to within a few lines of changes before a crash (or an accidental phone hang up). If you lose the contents of an editing buffer in a system crash, you will normally receive mail when you login that gives the name of the recovered file. To recover the file, enter the editor and type the command **recover (rec)**, followed by the name of the lost file. For example, to recover the buffer for an edit session involving the file "chap6", the command is:

```
:recover chap6
```

Recover is sometimes unable to save the entire buffer successfully, so always check the contents of the saved buffer carefully before writing it back onto the original file. For best results, write the buffer to a new file temporarily so you can examine it without risk to the original file. Unfortunately, you cannot use the recover command to retrieve a file you removed using the shell command **rm**.

Other recovery techniques

If something goes wrong when you are using the editor, it may be possible to save your work by using the command **preserve (pre)**, which saves the buffer as if the system had crashed. If you are writing a file and you get the message "Quota exceeded", you have tried to use more disk storage than is allotted to your account. *Proceed with caution* because it is likely that only a part of the editor's buffer is now present in the file you tried to write. In this case you should use the shell escape from the editor (!) to remove some files you don't need and try to write the file again. If this is not possible and you cannot find someone to help you, enter the command

```
:preserve
```



An Introduction to Display Editing with Vi

William Joy

Mark Horton

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, Ca. 94720

ABSTRACT

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 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.

The financial support of an IBM Graduate Fellowship and the National Science Foundation under grants MCS74-07644-A03 and MCS78-07291 is gratefully acknowledged.

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.‡

1.3. 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 l** keys as cursor positioning keys (these are labelled with arrows on an *adm3a*).*

(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.‡ 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.

‡ 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.

* 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.

‡ On smart terminals where it is possible, the editor will quietly flash the screen rather than ringing the bell.

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 `/`.†

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 `$`. Thus `/^searchCR` will search for the word 'search' at the beginning of a line, and `/last$CR` searches for the word 'last' at the end of a line.*

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.

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.

† 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 nowrapscanCR`, or more briefly `:se nowrapCR`.

*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 nomagicCR`; by putting this command in *EXINIT* in your environment, you can have this always be in effect (more about *EXINIT* later.)

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.† 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 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).

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

3.6. Summary

SPACE	advance the cursor one position
<code>^H</code>	backspace the cursor
<code>^W</code>	erase a word during an insert
erase	your erase (usually <code>^H</code> or <code>#</code>), erases a character during an insert
kill	your kill (usually <code>@</code> , <code>^X</code> , or <code>^U</code>), kills the insert on this line
.	repeats the changing command
<code>O</code>	opens and inputs new lines, above the current
<code>U</code>	undoes the changes you made to the current line
<code>a</code>	appends text after the cursor
<code>c</code>	changes the object you specify to the following text
<code>d</code>	deletes the object you specify
<code>i</code>	inserts text before the cursor
<code>o</code>	opens and inputs new lines, below the current
<code>u</code>	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 `fx` where `x` is this character. This command finds the next `x` character to the right of the cursor in the current line. Try then hitting a `,` which finds the next instance of the same character. By using the `f` command and then a sequence of `;`'s you can often get to a particular place in a line much faster than with a sequence of word motions or SPACES. There is also a `F` command, which is like `f`, but searches backward. The `;` command repeats `F` also.

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

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

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

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

"a5dd deleting 5 lines into the named buffer *a*. You can then move the cursor to the eventual resting place of these lines and do a "ap or "aP to put them back. In fact, you can switch and edit another file before you put the lines back, by giving a command of the form :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
fx	find <i>x</i> forward in line
p	put text back, after cursor or below current line
y	yank operator, for copies and moves
tx	up to <i>x</i> forward, for operators
Fx	f backward in line
P	put text back, before cursor or above current line
Tx	t backward in line

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 :wCR. 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 :wCR 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 :!cmdCR. 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.

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.†

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 7.8 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.

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

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

```
set opt=val
```

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

```
set opt
set noopt
```

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

You can get a list of all options which you have changed by the command `:setCR`, 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 nuCR`.

† Note that the command `z5.` has an entirely different effect, placing line 5 in the center of a new window.

`% 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. 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=10CR`. 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 aiCR`. 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=4CR` 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 `!)sortCR`. 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 lispCR`. 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

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 ^VBBBB
```

where B 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.†

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 nuCR** to enable this, and the command **:se nonuCR** to turn it off. You can have tabs represented as ^I and the ends of lines indicated with '\$' by giving the command **:se listCR**; **:se nolistCR** 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.

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:

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

of time you can give `:w` commands occasionally after major amounts of editing, and then finish with a `ZZ`. When you edit more than one file, you can finish with one 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 icCR`. The command `:se noicCR` 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.

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

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.‡

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.

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.

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



Ex Reference Manual

Version 3.7

William Joy

Mark Horton

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, Ca. 94720

ABSTRACT

Ex a line oriented text editor, which supports both command and display oriented editing. This reference manual describes the command oriented part of *ex*; the display editing features of *ex* are described in *An Introduction to Display Editing with Vi*. Other documents about the editor include the introduction *Edit: A tutorial*, the *Ex/edit Command Summary*, and a *Vi Quick Reference* card.

1. Starting *ex*

Each instance of the editor has a set of options, which can be set to tailor it to your liking. The command *edit* invokes a version of *ex* designed for more casual or beginning users by changing the default settings of some of these options. To simplify the description which follows we assume the default settings of the options.

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

A command to enter *ex* has the following prototype:†

```
ex [ - ] [ -v ] [ -t tag ] [ -r ] [ -l ] [ -wn ] [ -x ] [ -R ] [ +command ] name ...
```

The most common case edits a single file with no options, i.e.:

```
ex name
```

The *-* command line option suppresses all interactive-user feedback and is useful in processing editor scripts in command files. The *-v* option is equivalent to using *vi* rather than *ex*. The *-t* option is equivalent to an initial *tag* command, editing the file containing the *tag* and positioning the editor at its definition. The *-r* option is used in recovering after an

The financial support of an IBM Graduate Fellowship and the National Science Foundation under grants MCS74-07644-A03 and MCS78-07291 is gratefully acknowledged.

† Brackets '[' ']' surround optional parameters here.

2.5. Read only

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

3. Exceptional Conditions

3.1. Errors and interrupts

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

3.2. Recovering from hangups and crashes

If a hangup signal is received and the buffer has been modified since it was last written out, or if the system crashes, either the editor (in the first case) or the system (after it reboots in the second) will attempt to preserve the buffer. The next time you log in you should be able to recover the work you were doing, losing at most a few lines of changes from the last point before the hangup or editor crash. To recover a file you can use the `-r` option. If you were editing the file *resume*, then you should change to the directory where you were when the crash occurred, giving the command

```
ex -r resume
```

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

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

```
ex -r
```

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

4. Editing modes

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

The last three modes are *open* and *visual* modes, entered by the commands of the same name, and, within open and visual modes *text insertion* mode. *Open* and *visual* modes allow local editing operations to be performed on the text in the file. The *open* command displays one line at a time on any terminal while *visual* works on CRT terminals with random positioning cursors, using the screen as a (single) window for file editing changes. These modes are described (only) in *An Introduction to Display Editing with Vi*.

6. Command addressing

6.1. Addressing primitives

	The current line. Most commands leave the current line as the last line which they affect. The default address for most commands is the current line, thus '.' is rarely used alone as an address.
<i>n</i>	The <i>n</i> th line in the editor's buffer, lines being numbered sequentially from 1.
\$	The last line in the buffer.
%	An abbreviation for "1,\$", the entire buffer.
+ <i>n</i> - <i>n</i>	An offset relative to the current buffer line.†
/pat/ ?pat?	Scan forward and backward respectively for a line containing <i>pat</i> , a regular expression (as defined below). The scans normally wrap around the end of the buffer. If all that is desired is to print the next line containing <i>pat</i> , then the trailing / or ? may be omitted. If <i>pat</i> is omitted or explicitly empty, then the last regular expression specified is located.‡
~x	Before each non-relative motion of the current line '.', the previous current line is marked with a tag, subsequently referred to as '~'. This makes it easy to refer or return to this previous context. Marks may also be established by the <i>mark</i> command, using single lower case letters <i>x</i> and the marked lines referred to as "x".

6.2. Combining addressing primitives

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

7. Command descriptions

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

address command ! parameters count flags

All parts are optional; the degenerate case is the empty command which prints the next line in the file. For sanity with use from within *visual* mode, *ex* ignores a ":" preceding any command.

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

abbreviate *word rhs*

abbr: *ab*

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

† The forms '+3' '+3' and '+++' are all equivalent; if the current line is line 100 they all address line 103.

‡ The forms \ and \? scan using the last regular expression used in a scan; after a substitute // and ?? would scan using the substitute's regular expression.

† Null address specifications are permitted in a list of addresses, the default in this case is the current line '.'; thus ',100' is equivalent to ',.100'. It is an error to give a prefix address to a command which expects none.

trailing newline character, it will be supplied and a complaint will be issued. This command leaves the current line ‘.’ at the last line read.‡

e! *file*

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

e +n *file*

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

file

abbr: f

Prints the current file name, whether it has been ‘[Modified]’ since the last *write* command, whether it is *read only*, the current line, the number of lines in the buffer, and the percentage of the way through the buffer of the current line.*

file *file*

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

(1 , \$) global /pat /cmds

abbr: g

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

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

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

g! /pat /cmds

abbr: v

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

(.) insert

abbr: i

text

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

‡ If executed from within *open* or *visual*, the current line is initially the first line of the file.

* In the rare case that the current file is ‘[Not edited]’ this is noted also; in this case you have to use the form **w!** to write to the file, since the editor is not sure that a *write* will not destroy a file unrelated to the current contents of the buffer.

n *filelist***n** +*command filelist*

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

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

Prints each specified line preceded by its buffer line number. The current line is left at the last line printed.

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

Enters intraline editing *open* mode at each addressed line. If *pat* is given, then the cursor will be placed initially at the beginning of the string matched by the pattern. To exit this mode use Q. See *An Introduction to Display Editing with Vi* for more details.

preserve

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

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

Prints the specified lines with non-printing characters printed as control characters "x"; delete (octal 177) is represented as "?". The current line is left at the last line printed.

(.) *put buffer* abbr: pu

Puts back previously *deleted* or *yanked* lines. Normally used with *delete* to effect movement of lines, or with *yank* to effect duplication of lines. If no *buffer* is specified, then the last *deleted* or *yanked* text is restored.* By using a named buffer, text may be restored that was saved there at any previous time.

quit abbr: q

Causes *ex* to terminate. No automatic write of the editor buffer to a file is performed. However, *ex* issues a warning message if the file has changed since the last *write* command was issued, and does not *quit*.† Normally, you will wish to save your changes, and you should give a *write* command; if you wish to discard them, use the *q!* command variant.

q!

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

(.) *read file* abbr: r

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

* But no modifying commands may intervene between the *delete* or *yank* and the *put*, nor may lines be moved between files without using a named buffer.

† *Ex* will also issue a diagnostic if there are more files in the argument list.

stop

Suspends the editor, returning control to the top level shell. If *autowrite* is set and there are unsaved changes, a write is done first unless the form **stop!** is used. This command is only available where supported by the teletype driver and operating system.

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

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

(. . .) **t** *addr flags*

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

ta tag

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

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

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

unabbreviate *word* abbr: una

Delete *word* from the list of abbreviations.

undo abbr: u

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

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

unmap lhs

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

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

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

version abbr: ve

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

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

(. +1) z count

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

(.) z type count

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

! command

The remainder of the line after the '!' character is sent to a shell to be executed. Within the text of *command* the characters '%' and '#' are expanded as in filenames and the character '!' is replaced with the text of the previous command. Thus, in particular, '!!' repeats the last such shell escape. If any such expansion is performed, the expanded line will be echoed. The current line is unchanged by this command.

If there has been "[No write]" of the buffer contents since the last change to the editing buffer, then a diagnostic will be printed before the command is executed as a warning. A single '!' is printed when the command completes.

(addr , addr) ! command

Takes the specified address range and supplies it as standard input to *command*; the resulting output then replaces the input lines.

(\$) =

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

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

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

^D

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

(. +1 , +1)**(. +1 , +1) |**

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

(. , .) & options count flags

Repeats the previous *substitute* command.

* Forms 'z=' and 'z*' also exist; 'z=' places the current line in the center, surrounds it with lines of '-' characters and leaves the current line at this line. The form 'z*' prints the window before 'z-' would. The characters '+', '*' and '-' may be repeated for cumulative effect. On some v2 editors, no *type* may be given.

letter. If the first character of *string* is an '^' then the construct matches those characters which it otherwise would not; thus '[^a-z]' matches anything but a lower-case letter (and of course a newline). To place any of the characters '^', '[', or '-' in *string* you must escape them with a preceding '\'

8.4. Combining regular expression primitives

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

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

8.5. Substitute replacement patterns

The basic metacharacters for the replacement pattern are '&' and '~'; these are given as '\&' and '\~' when *nomagic* is set. Each instance of '&' is replaced by the characters which the regular expression matched. The metacharacter '~' stands, in the replacement pattern, for the defining text of the previous replacement pattern.

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

9. Option descriptions

autoindent, ai

default: noai

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

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

Specially processed in this mode is a line with no characters added to it, which turns into a completely blank line (the white space provided for the *autoindent* is discarded.) Also specially processed in this mode are lines beginning with an '^' and immediately followed by a ^D. This causes the input to be repositioned at the beginning of the line, but retaining the previous indent for the next line. Similarly, a '0' followed by a ^D

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

- list** default: nolist
 All printed lines will be displayed (more) unambiguously, showing tabs and end-of-lines as in the *list* command.
- magic** default: magic for *ex* and *vi*†
 If *nomagic* is set, the number of regular expression metacharacters is greatly reduced, with only '^' and '\$' having special effects. In addition the metacharacters '~' and '&' of the replacement pattern are treated as normal characters. All the normal metacharacters may be made *magic* when *nomagic* is set by preceding them with a '\.'
- mesg** default: mesg
 Causes write permission to be turned off to the terminal while you are in visual mode, if *nomesg* is set.
- modeline** default: nomodeline
 If *modeline* is set, then the first 5 lines and the last five lines of the file will be checked for *ex* command lines and the commands issued. To be recognized as a command line, the line must have the string *ex:* or *vi:* preceded by a tab or a space. This string may be anywhere in the line and anything after the *:* is interpreted as editor commands. This option defaults to off because of unexpected behavior when editing files such as *etc/passwd*.
- number, nu** default: nonumber
 Causes all output lines to be printed with their line numbers. In addition each input line will be prompted for by supplying the line number it will have.
- open** default: open
 If *noopen*, the commands *open* and *visual* are not permitted. This is set for *edit* to prevent confusion resulting from accidental entry to open or visual mode.
- optimize, opt** default: optimize
 Throughput of text is expedited by setting the terminal to not do automatic carriage returns when printing more than one (logical) line of output, greatly speeding output on terminals without addressable cursors when text with leading white space is printed.
- paragraphs, para** default: para=IPLPPPQPP LIbp
 Specifies the paragraphs for the { and } operations in *open* and *visual*. The pairs of characters in the option's value are the names of the macros which start paragraphs.
- prompt** default: prompt
 Command mode input is prompted for with a ':
- redraw** default: noredraw
 The editor simulates (using great amounts of output), an intelligent terminal on a dumb terminal (e.g. during insertions in *visual* the characters to the right of the cursor position are refreshed as each input character is typed.) Useful only at very high speed.

† *Nomagic* for *edit*.

- tags** default: tags=tags /usr/lib/tags
A path of files to be used as tag files for the *tag* command. A requested tag is searched for in the specified files, sequentially. By default, files called **tags** are searched for in the current directory and in /usr/lib (a master file for the entire system).
- term** from environment TERM
The terminal type of the output device.
- terse** default: noterse
Shorter error diagnostics are produced for the experienced user.
- warn** default: warn
Warn if there has been '[No write since last change]' before a '!' command escape.
- window** default: window=speed dependent
The number of lines in a text window in the *visual* command. The default is 8 at slow speeds (600 baud or less), 16 at medium speed (1200 baud), and the full screen (minus one line) at higher speeds.
- w300, w1200, w9600**
These are not true options but set **window** only if the speed is slow (300), medium (1200), or high (9600), respectively. They are suitable for an EXINIT and make it easy to change the 8/16/full screen rule.
- wrapsan, ws** default: ws
Searches using the regular expressions in addressing will wrap around past the end of the file.
- wrapmargin, wm** default: wm=0
Defines a margin for automatic wrapover of text during input in *open* and *visual* modes. See *An Introduction to Text Editing with Vi* for details.
- writeany, wa** default: nowa
Inhibit the checks normally made before *write* commands, allowing a write to any file which the system protection mechanism will allow.

10. Limitations

Editor limits that the user is likely to encounter are as follows: 1024 characters per line, 256 characters per global command list, 128 characters per file name, 128 characters in the previous inserted and deleted text in *open* or *visual*, 100 characters in a shell escape command, 63 characters in a string valued option, and 30 characters in a tag name, and a limit of 250000 lines in the file is silently enforced.

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

Acknowledgments. Chuck Haley contributed greatly to the early development of *ex*. Bruce Englar encouraged the redesign which led to *ex* version 1. 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.

JOVE Manual for UNIX Users

Jonathan Payne

(revised for 4.3BSD by Doug Kingston and Mark Seiden)

1. Introduction

JOVE* is an advanced, self-documenting, customizable real-time display editor. It (and this tutorial introduction) are based on the original EMACS editor and user manual written at M.I.T. by Richard Stallman+.

JOVE is considered a *display* editor because normally the text being edited is visible on the screen and is updated automatically as you type your commands.

It's considered a *real-time* editor because the display is updated very frequently, usually after each character or pair of characters you type. This minimizes the amount of information you must keep in your head as you edit.

JOVE is *advanced* because it provides facilities that go beyond simple insertion and deletion: filling of text; automatic indentations of programs; view more than one file at once; and dealing in terms of characters, words, lines, sentences and paragraphs. It is much easier to type one command meaning "go to the end of the paragraph" than to find the desired spot with repetition of simpler commands.

Self-documenting means that at almost any time you can easily find out what a command does, or to find all the commands that pertain to a topic.

Customizable means that you can change the definition of JOVE commands in little ways. For example, you can rearrange the command set; if you prefer to use arrow keys for the four basic cursor motion commands (up, down, left and right), you can. Another sort of customization is writing new commands by combining built in commands.

2. The Organization of the Screen

JOVE divides the screen up into several sections. The biggest of these sections is used to display the text you are editing. The terminal's cursor shows the position of *point*, the location at which editing takes place. While the cursor appears to point *at* a character, *point* should be thought of as between characters; it points *before* the character that the cursor appears on top of. Terminals have only one cursor, and when output is in progress it must appear where the typing is being done. This doesn't mean that *point* is moving; it is only that JOVE has no way of showing you the location of *point* except when the terminal is idle.

The lines of the screen are usually available for displaying text but sometimes are pre-empted by timeout from certain commands (such as a listing of all the editor commands). Most of the time, output from commands like these is only desired for a short period of time, usually just long enough to glance at it. When you have finished looking at the output, you can type Space to make your text reappear. (Usually a Space that you type inserts itself, but when there is timeout on the screen, it does nothing but get rid of that). Any other command executes normally, *after* redrawing your text.

2.1. The Message Line

The bottom line on the screen, called the *message line*, is reserved for printing messages and for accepting input from the user, such as filenames or search strings. When JOVE prompts for input, the cursor will temporarily appear on the bottom line, waiting for you to type a string. When you have finished typing your input, you can type a Return to send it to JOVE. If you change your mind about running the command that is waiting for input, you can type Control-G to abort, and you can

*JOVE stands for Jonathan's Own Version of Emacs.

+Although JOVE is meant to be compatible with EMACS, and indeed many of the basic commands are very similar, there are some major differences between the two editors, and you should not rely on their behaving identically.

see a list of all the commands whose names begin with the characters you've already typed; you can type Space to have JOVE supply as many characters as it can; or you can type Return to complete the command if there is only one possibility. For example, if you have typed the letters "au" and you then type a question mark, you will see the list

```
auto-execute-command
auto-execute-macro
auto-fill-mode
auto-indent-mode
```

If you type a Return at this point, JOVE will complain by ringing the bell, because the letters you've typed do not unambiguously specify a single command. But if you type Space, JOVE will supply the characters "to-" because all commands that begin "au" also begin "auto-". You could then type the letter "f" followed by either Space or Return, and JOVE would complete the entire command.

Whenever JOVE is prompting you for a filename, say in the *find-file* command, you also need only type enough of the name to make it unambiguous with respect to files that already exist. In this case, question mark and Space work just as they do in command completion, but Return always accepts the name just as you've typed it, because you might want to create a new file with a name similar to that of an existing file.

4. Commands and Variables

JOVE is composed of *commands* which have long names such as *next-line*. Then *keys* such as C-N are connected to commands through the *command dispatch table*. When we say that C-N moves the cursor down a line, we are glossing over a distinction which is unimportant for ordinary use, but essential for simple customization: it is the command *next-line* which knows how to move a down line, and C-N moves down a line because it is connected to that command. The name for this connection is a *binding*; we say that the key C-N is *bound to* the command *next-line*.

Not all commands are bound to keys. To invoke a command that isn't bound to a key, you can type the sequence ESC X, which is bound to the command *execute-named-command*. You will then be able to type the name of whatever command you want to execute on the message line.

Sometimes the description of a command will say "to change this, set the variable *mumble-foo*". A variable is a name used to remember a value. JOVE contains variables which are there so that you can change them if you want to customize. The variable's value is examined by some command, and changing that value makes the command behave differently. Until you are interesting in customizing JOVE, you can ignore this information.

4.1. Prefix Characters

Because there are more command names than keys, JOVE provides *prefix characters* to increase the number of commands that can be invoked quickly and easily. When you type a prefix character JOVE will wait for another character before deciding what to do. If you wait more than a second or so, JOVE will print the prefix character on the message line as a reminder and leave the cursor down there until you type your next character. There are two prefix characters built into JOVE: Escape and Control-X. How the next character is interpreted depends on which prefix character you typed. For example, if you type Escape followed by B you'll run *backward-word*, but if you type Control-X followed by B you'll run *select-buffer*. Elsewhere in this manual, the Escape key is indicated as "ESC", which is also what JOVE displays on the message line for Escape.

4.2. Help

To get a list of keys and their associated commands, you type ESC X *describe-bindings*. If you want to describe a single key, ESC X *describe-key* will work. A description of an individual command is available by using ESC X *describe-command*, and descriptions of variables by using ESC X *describe-variable*. If you can't remember the name of the thing you want to know about, ESC X *apropos* will tell you if a command or variable has a given string in its name. For example, ESC X *apropos describe* will list the names of the four describe commands mentioned briefly in this section.

5.3. Erasing Text

Rubout	Delete the character before the cursor.
C-D	Delete the character after the cursor.
C-K	Kill to the end of the line.

You already know about the Rubout command which deletes the character before the cursor. Another command, Control-D, deletes the character after the cursor, causing the rest of the text on the line to shift left. If Control-D is typed at the end of a line, that line and the next line are joined together.

To erase a larger amount of text, use the Control-K command, which kills a line at a time. If Control-K is done at the beginning or middle of a line, it kills all the text up to the end of the line. If Control-K is done at the end of a line, it joins that line and the next line. If Control-K is done twice, it kills the rest of the line and the line separator also.

5.4. Files — Saving Your Work

The commands above are sufficient for creating text in the JOVE buffer. The more advanced JOVE commands just make things easier. But to keep any text permanently you must put it in a *file*. Files are the objects which UNIX† uses for storing data for a length of time. To tell JOVE to read text into a file, choose a filename, such as *foo.bar*, and type C-X C-R *foo.bar*<return>. This reads the file *foo.bar* so that its contents appear on the screen for editing. You can make changes, and then save the file by typing C-X C-S (save-file). This makes the changes permanent and actually changes the file *foo.bar*. Until then, the changes are only inside JOVE, and the file *foo.bar* is not really changed. If the file *foo.bar* doesn't exist, and you want to create it, read it as if it did exist. When you save your text with C-X C-S the file will be created.

5.5. Exiting and Pausing — Leaving JOVE

The command C-X C-C (*exit-jove*) will terminate the JOVE session and return to the shell. If there are modified but unsaved buffers, JOVE will ask you for confirmation, and you can abort the command, look at what buffers are modified but unsaved using C-X C-B (*list-buffers*), save the valuable ones, and then exit. If what you want to do, on the other hand, is *preserve* the editing session but return to the shell temporarily you can (under Berkeley UNIX only) issue the command ESC S (*pause-jove*), do your UNIX work within the c-shell, then return to JOVE using the *fg* command to resume editing at the point where you paused. For this sort of situation you might consider using an *interactive shell* (that is, a shell in a JOVE window) which lets you use editor commands to manipulate your UNIX commands (and their output) while never leaving the editor. (The interactive shell feature is described below.)

5.6. Giving Numeric Arguments to JOVE Commands

Any JOVE command can be given a *numeric argument*. Some commands interpret the argument as a repetition count. For example, giving an argument of ten to the C-F command (forward-character) moves forward ten characters. With these commands, no argument is equivalent to an argument of 1.

Some commands use the value of the argument, but do something peculiar (or nothing) when there is no argument. For example, ESC G (*goto-line*) with an argument *n* goes to the beginning of the *n*'th line. But ESC G with no argument doesn't do anything. Similarly, C-K with an argument kills that many lines, including their line separators. Without an argument, C-K when there is text on the line to the right of the cursor kills that text; when there is no text after the cursor, C-K deletes the line separator.

The fundamental way of specifying an argument is to use ESC followed by the digits of the argument, for example, ESC 123 ESC G to go to line 123. Negative arguments are allowed, although not all of the commands know what to do with one.

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

5.9. Killing and Moving Text

The most common way of moving or copying text with JOVE is to kill it, and get it back again in one or more places. This is very safe because the last several pieces of killed text are all remembered, and it is versatile, because the many commands for killing syntactic units can also be used for moving those units. There are also other ways of moving text for special purposes.

5.10. Deletion and Killing

Most commands which erase text from the buffer save it so that you can get it back if you change your mind, or move or copy it to other parts of the buffer. These commands are known as *kill* commands. The rest of the commands that erase text do not save it; they are known as *delete* commands. The delete commands include C-D and Rubout, which delete only one character at a time, and those commands that delete only spaces or line separators. Commands that can destroy significant amounts of nontrivial data generally kill. A command's name and description will use the words *kill* or *delete* to say which one it does.

C-D	Delete next character.
Rubout	Delete previous character.
ESC \	Delete spaces and tabs around point.
C-X C-O	Delete blank lines around the current line.
C-K	Kill rest of line or one or more lines.
C-W	Kill region (from point to the mark).
ESC D	Kill word.
ESC Rubout	Kill word backwards.
ESC K	Kill to end of sentence.
C-X Rubout	Kill to beginning of sentence.

5.11. Deletion

The most basic delete commands are C-D and Rubout. C-D deletes the character after the cursor, the one the cursor is "on top of" or "underneath". The cursor doesn't move. Rubout deletes the character before the cursor, and moves the cursor back. Line separators act like normal characters when deleted. Actually, C-D and Rubout aren't always *delete* commands; if you give an argument, they *kill* instead. This prevents you from losing a great deal of text by typing a large argument to a C-D or Rubout.

The other delete commands are those which delete only formatting characters: spaces, tabs, and line separators. ESC \ (*delete-white-space*) deletes all the spaces and tab characters before and after point. C-X C-O (*delete-blank-lines*) deletes all blank lines after the current line, and if the current line is blank deletes all the blank lines preceding the current line as well (leaving one blank line, the current line).

5.12. Killing by Lines

The simplest kill command is the C-K command. If issued at the beginning of a line, it kills all the text on the line, leaving it blank. If given on a line containing only white space (blanks and tabs) the line disappears. As a consequence, if you go to the front of a non-blank line and type two C-K's, the line disappears completely.

More generally, C-K kills from point up to the end of the line, unless it is at the end of a line. In that case, it kills the line separator following the line, thus merging the next line into the current one. Invisible spaces and tabs at the end of the line are ignored when deciding which case applies, so if point appears to be at the end of the line, you can be sure the line separator will be killed.

C-K with an argument of zero kills all the text before point on the current line.

When the text you are looking for is brought into the buffer, you can stop doing ESC Y's and the text will stay there. It's really just a copy of what's at the front of the ring, so editing it does not change what's in the ring. And the ring, once rotated, stays rotated, so that doing another C-Y gets another copy of what you rotated to the front with ESC Y.

If you change your mind about un-killing, C-W gets rid of the un-killed text, even after any number of ESC Y's.

6. Searching

The search commands are useful for finding and moving to arbitrary positions in the buffer in one swift motion. For example, if you just ran the spell program on a paper and you want to correct some word, you can use the search commands to move directly to that word. There are two flavors of search: *string search* and *incremental search*. The former is the default flavor—if you want to use incremental search you must rearrange the key bindings (see below).

6.1. Conventional Search

C-S Search forward.

C-R Search backward.

To search for the string "FOO" you type "C-S FOO<return>". If JOVE finds FOO it moves point to the end of it; otherwise JOVE prints an error message and leaves point unchanged. C-S searches forward from point so only occurrences of FOO after point are found. To search in the other direction use C-R. It is exactly the same as C-S except it searches in the opposite direction, and if it finds the string, it leaves point at the beginning of it, not at the end as in C-S.

While JOVE is searching it prints the search string on the message line. This is so you know what JOVE is doing. When the system is heavily loaded and editing in exceptionally large buffers, searches can take several (sometimes many) seconds.

JOVE remembers the last search string you used, so if you want to search for the same string you can type "C-S <return>". If you mistyped the last search string, you can type C-S followed by C-R. C-R, as usual, inserts the default search string into the minibuffer, and then you can fix it up.

6.2. Incremental Search

This search command is unusual in that it is *incremental*; it begins to search before you have typed the complete search string. As you type in the search string, JOVE shows you where it would be found. When you have typed enough characters to identify the place you want, you can stop. Depending on what you will do next, you may or may not need to terminate the search explicitly with a Return first.

The command to search is C-S (*i-search-forward*). C-S reads in characters and positions the cursor at the first occurrence of the characters that you have typed so far. If you type C-S and then F, the cursor moves in the text just after the next "F". Type an "O", and see the cursor move to after the next "FO". After another "O", the cursor is after the next "FOO". At the same time, the "FOO" has echoed on the message line.

If you type a mistaken character, you can rub it out. After the FOO, typing a Rubout makes the "O" disappear from the message line, leaving only "FO". The cursor moves back in the buffer to the "FO". Rubbing out the "O" and "F" moves the cursor back to where you started the search.

When you are satisfied with the place you have reached, you can type a Return, which stops searching, leaving the cursor where the search brought it. Also, any command not specially meaningful in searches stops the searching and is then executed. Thus, typing C-A would exit the search and then move to the beginning of the line. Return is necessary only if the next character you want to type is a printing character, Rubout, Return, or another search command, since those are the characters that have special meanings inside the search.

Sometimes you search for "FOO" and find it, but not the one you hoped to find. Perhaps there is a second FOO that you forgot about, after the one you just found. Then type another C-S and the

Rubout	to skip to the next FOO without replacing this one.
Return	to stop without doing any more replacements.
Period	to replace this FOO and then stop.
! or P	to replace all remaining FOO's without asking.
C-R or R	to enter a recursive editing level, in case the FOO needs to be edited rather than just replaced with a BAR. When you are done, exit the recursive editing level with C-X C-C and the next FOO will be displayed.
C-W	to delete the FOO, and then start editing the buffer. When you are finished editing whatever is to replace the FOO, exit the recursive editing level with C-X C-C and the next FOO will be displayed.
U	move to the last replacement and undo it.

Another alternative is using *replace-in-region* which is just like *replace-string* except it searches only within the region.

8. Commands for English Text

JOVE has many commands that work on the basic units of English text: words, sentences and paragraphs.

8.1. Word Commands

JOVE has commands for moving over or operating on words. By convention, they are all ESC commands.

ESC F	Move Forward over a word.
ESC B	Move Backward over a word.
ESC D	Kill forward to the end of a word.
ESC Rubout	Kill backward to the beginning of a word.

Notice how these commands form a group that parallels the character-based commands, C-F, C-B, C-D, and Rubout.

The commands ESC F and ESC B move forward and backward over words. They are thus analogous to Control-F and Control-B, which move over single characters. Like their Control- analogues, ESC F and ESC B move several words if given an argument. ESC F with a negative argument moves backward like ESC B, and ESC B with a negative argument moves forward. Forward motion stops right after the last letter of the word, while backward motion stops right before the first letter.

It is easy to kill a word at a time. ESC D kills the word after point. To be precise, it kills everything from point to the place ESC F would move to. Thus, if point is in the middle of a word, only the part after point is killed. If some punctuation comes after point, and before the next word, it is killed along with the word. If you wish to kill only the next word but not the punctuation, simply do ESC F to get to the end, and kill the word backwards with ESC Rubout. ESC D takes arguments just like ESC F.

ESC Rubout kills the word before point. It kills everything from point back to where ESC B would move to. If point is after the space in "FOO, BAR", then "FOO, " is killed. If you wish to kill just "FOO", then do a ESC B and a ESC D instead of a ESC Rubout.

8.2. Sentence Commands

The JOVE commands for manipulating sentences and paragraphs are mostly ESC commands, so as to resemble the word-handling commands.

ESC A	Move back to the beginning of the sentence.
-------	---

Normally ESC J figures out the indent of the paragraph and uses that same indent when filling. If you want to change the indent of a paragraph you set *left-margin* to the new position and type C-U ESC J. *fill-paragraph*, when supplied a numeric argument, uses the value of *left-margin*.

If you know where you want to set the right margin but you don't know the actual value, move to where you want to set the value and use the *right-margin-here* command. *left-margin-here* does the same for the *left-margin* variable.

8.6. Case Conversion Commands

ESC L	Convert following word to lower case.
ESC U	Convert following word to upper case.
ESC C	Capitalize the following word.

The word conversion commands are most useful. ESC L converts the word after point to lower case, moving past it. Thus, successive ESC L's convert successive words. ESC U converts to all capitals instead, while ESC C puts the first letter of the word into upper case and the rest into lower case. All these commands convert several words at once if given an argument. They are especially convenient for converting a large amount of text from all upper case to mixed case, because you can move through the text using ESC L, ESC U or ESC C on each word as appropriate.

When given a negative argument, the word case conversion commands apply to the appropriate number of words before point, but do not move point. This is convenient when you have just typed a word in the wrong case. You can give the case conversion command and continue typing.

If a word case conversion command is given in the middle of a word, it applies only to the part of the word which follows the cursor, treating it as a whole word.

The other case conversion functions are *case-region-upper* and *case-region-lower*, which convert everything between point and mark to the specified case. Point and mark remain unchanged.

8.7. Commands for Fixing Typos

In this section we describe the commands that are especially useful for the times when you catch a mistake on your text after you have made it, or change your mind while composing text on line.

Rubout	Delete last character.
ESC Rubout	Kill last word.
C-X Rubout	Kill to beginning of sentence.
C-T	Transpose two characters.
C-X C-T	Transpose two lines.
ESC Minus ESC L	Convert last word to lower case.
ESC Minus ESC U	Convert last word to upper case.
ESC Minus ESC C	Convert last word to lower case with capital initial.

8.8. Killing Your Mistakes

The Rubout command is the most important correction command. When used among printing (self-inserting) characters, it can be thought of as canceling the last character typed.

When your mistake is longer than a couple of characters, it might be more convenient to use ESC Rubout or C-X Rubout. ESC Rubout kills back to the start of the last word, and C-X Rubout kills back to the start of the last sentence. C-X Rubout is particularly useful when you are thinking of what to write as you type it, in case you change your mind about phrasing. ESC Rubout and C-X Rubout save the killed text for C-Y and ESC Y to retrieve.

ESC Rubout is often useful even when you have typed only a few characters wrong, if you know you are confused in your typing and aren't sure what you typed. At such a time, you cannot correct with Rubout except by looking at the screen to see what you did. It requires less thought to kill the whole

If you wish to save the file and make your changes permanent, type C-X C-S. After the save is finished, C-X C-S prints the filename and the number of characters and lines that it wrote to the file. If there are no changes to save (no asterisk at the end of the mode line), the file is not saved; otherwise the changes saved and the asterisk at the end of the mode line will disappear.

What if you want to create a file? Just visit it. JOVE prints (*New file*) but aside from that behaves as if you had visited an existing empty file. If you make any changes and save them, the file is created. If you visit a nonexistent file unintentionally (because you typed the wrong filename), go ahead and visit the file you meant. If you don't save the unwanted file, it is not created.

If you alter one file and then visit another in the same buffer, JOVE offers to save the old one. If you answer YES, the old file is saved; if you answer NO, all the changes you have made to it since the last save are lost. You should not type ahead after a file visiting command, because your type-ahead might answer an unexpected question in a way that you would regret.

Sometimes you will change a buffer by accident. Even if you undo the effect of the change by editing, JOVE still knows that "the buffer has been changed". You can tell JOVE to pretend that there have been no changes with the ESC ~ command (*make-buffer-unmodified*). This command simply clears the "modified" flag which says that the buffer contains changes which need to be saved. Even if the buffer really *is* changed JOVE will still act as if it were not.

If JOVE is about to save a file and sees that the date of the version on disk does not match what JOVE last read or wrote, JOVE notifies you of this fact, and asks what to do, because this probably means that something is wrong. For example, somebody else may have been editing the same file. If this is so, there is a good chance that your work or his work will be lost if you don't take the proper steps. You should first find out exactly what is going on. If you determine that somebody else has modified the file, save your file under a different filename and then DIFF the two files to merge the two sets of changes. (The "patch" command is useful for applying the results of context diffs directly). Also get in touch with the other person so that the files don't diverge any further.

9.2. How to Undo Drastic Changes to a File

If you have made several extensive changes to a file and then change your mind about them, and you haven't yet saved them, you can get rid of them by reading in the previous version of the file. You can do this with the C-X C-V command, to visit the unsaved version of the file.

9.3. Recovering from system/editor crashes

JOVE does not have *Auto Save* mode, but it does provide a way to recover your work in the event of a system or editor crash. JOVE saves information about the files you're editing every so many changes to a buffer to make recovery possible. Since a relatively small amount of information is involved it's hardly even noticeable when JOVE does this. The variable "sync-frequency" says how often to save the necessary information, and the default is every 50 changes. 50 is a very reasonable number: if you are writing a paper you will not lose more than the last 50 characters you typed, which is less than the average length of a line.

9.4. Miscellaneous File Operations

ESC X *write-file* <file><return> writes the contents of the buffer into the file <file>, and then visits that file. It can be thought of as a way of "changing the name" of the file you are visiting. Unlike C-X C-S, *write-file* saves even if the buffer has not been changed. C-X C-W is another way of getting this command.

ESC X *insert-file* <file><return> inserts the contents of <file> into the buffer at point, leaving point unchanged before the contents. You can also use C-X C-I to get this command.

ESC X *write-region* <file><return> writes the region (the text between point and mark) to the specified file. It does not set the visited filename. The buffer is not changed.

ESC X *append-region* <file><return> appends the region to <file>. The text is added to the end of <file>.

error. When this happens you will want to kill some buffers with the C-X K (*delete-buffer*) command. You can kill the buffer FOO by doing C-X K FOO<return>. If you type C-X K <return> JOVE will kill the previously selected buffer. If you try to kill a buffer that needs saving JOVE will ask you to confirm it.

If you need to kill several buffers, use the command *kill-some-buffers*. This prompts you with the name of each buffer and asks for confirmation before killing that buffer.

11. Controlling the Display

Since only part of a large file will fit on the screen, JOVE tries to show the part that is likely to be interesting. The display control commands allow you to see a different part of the file.

- C-L Reposition point at a specified vertical position, OR clear and redraw the screen with point in the same place.
- C-V Scroll forwards (a screen or a few lines).
- ESC V Scroll backwards.
- C-Z Scroll forward some lines.
- ESC Z Scroll backwards some lines.

The terminal screen is rarely large enough to display all of your file. If the whole buffer doesn't fit on the screen, JOVE shows a contiguous portion of it, containing *point*. It continues to show approximately the same portion until point moves outside of what is displayed; then JOVE chooses a new portion centered around the new *point*. This is JOVE's guess as to what you are most interested in seeing, but if the guess is wrong, you can use the display control commands to see a different portion. The available screen area through which you can see part of the buffer is called *the window*, and the choice of where in the buffer to start displaying is also called *the window*. (When there is only one window, it plus the mode line and the input line take up the whole screen).

First we describe how JOVE chooses a new window position on its own. The goal is usually to place *point* half way down the window. This is controlled by the variable *scroll-step*, whose value is the number of lines above the bottom or below the top of the window that the line containing point is placed. A value of 0 (the initial value) means center *point* in the window.

The basic display control command is C-L (*redraw-display*). In its simplest form, with no argument, it tells JOVE to choose a new window position, centering point half way from the top as usual.

C-L with a positive argument chooses a new window so as to put point that many lines from the top. An argument of zero puts point on the very top line. Point does not move with respect to the text; rather, the text and point move rigidly on the screen.

If point stays on the same line, the window is first cleared and then redrawn. Thus, two C-L's in a row are guaranteed to clear the current window. ESC C-L will clear and redraw the entire screen.

The *scrolling* commands C-V, ESC V, C-Z, and ESC Z, let you move the whole display up or down a few lines. C-V (*next-page*) with an argument shows you that many more lines at the bottom of the screen, moving the text and point up together as C-L might. C-V with a negative argument shows you more lines at the top of the screen, as does ESC V (*previous-page*) with a positive argument.

To read the buffer a window at a time, use the C-V command with no argument. It takes the last line at the bottom of the window and puts it at the top, followed by nearly a whole window of lines not visible before. Point is put at the top of the window. Thus, each C-V shows the "next page of text", except for one line of overlap to provide context. To move backward, use ESC V without an argument, which moves a whole window backwards (again with a line of overlap).

C-Z and ESC Z scroll one line forward and one line backward, respectively. These are convenient for moving in units of lines without having to type a numeric argument.

If you have the same buffer in both windows, you must beware of trying to visit a different file in one of the windows with C-X C-V, because if you bring a new file into this buffer, it will replace the old file in *both* windows. To view different files in different windows, you must switch buffers in one of the windows first (with C-X B or C-X C-F, perhaps).

A convenient "combination" command for viewing something in another window is C-X 4 (*window-find*). With this command you can ask to see any specified buffer, file or tag in the other window. Follow the C-X 4 with either B and a buffer name, F and a filename, or T and a tag name. This switches to the other window and finds there what you specified. If you were previously in one-window mode, multiple-window mode is entered. C-X 4 B is similar to C-X 2 C-X B. C-X 4 F is similar to C-X 2 C-X C-F. C-X 4 T is similar to C-X 2 C-X T. The difference is one of efficiency, and also that C-X 4 works equally well if you are already using two windows.

12. Processes Under JOVE

Another feature in JOVE is its ability to interact with UNIX in a useful way. You can run other UNIX commands from JOVE and catch their output in JOVE buffers. In this chapter we will discuss the different ways to run and interact with UNIX commands.

12.1. Non-interactive UNIX commands

To run a UNIX command from JOVE just type "C-X !" followed by the name of the command terminated with Return. For example, to get a list of all the users on the system, you do:

```
C-X ! who<return>
```

Then JOVE picks a reasonable buffer in which the output from the command will be placed. E.g., "who" uses a buffer called *who*; "ps alx" uses *ps*; and "fgrep -n foo *.c" uses *fgrep*. If JOVE wants to use a buffer that already exists it first erases the old contents. If the buffer it selects holds a file, not output from a previous shell command, you must first delete that buffer with C-X K.

Once JOVE has picked a buffer it puts that buffer in a window so you can see the command's output as it is running. If there is only one window JOVE will automatically make another one. Otherwise, JOVE tries to pick the most convenient window which isn't the current one.

It's not a good idea to type anything while the command is running. There are two reasons for this:

- (i) JOVE won't see the characters (thus won't execute them) until the command finishes, so you may forget what you've typed.
- (ii) Although JOVE won't know what you've typed, it *will* know that you've typed something, and then it will try to be "smart" and not update the display until it's interpreted what you've typed. But, of course, JOVE won't interpret what you type until the UNIX command completes, so you're left with the uneasy feeling you get when you don't know what the hell the computer is doing*.

If you want to interrupt the command for some reason (perhaps you mistyped it, or you changed your mind) you can type C-]. Typing this inside JOVE while a process is running is the same as typing C-C when you are outside JOVE, namely the process stops in a hurry.

When the command finishes, JOVE puts you back in the window in which you started. Then it prints a message indicating whether or not the command completed successfully in its (the command's) opinion. That is, if the command had what it considers an error (or you interrupt it with C-]) JOVE will print an appropriate message.

*This is a bug and should be fixed, but probably won't be for a while.

type ESC X *list-processes*<return> to get a list of each process and its state. If your process died abnormally, *list-processes* may help you figure out why.

12.6. How to Run a Shell in a Window

Type ESC X *i-shell*<return> to start up a shell. As with C-X !, JOVE will create a buffer, called *shell-1*, and select a window for this new buffer. But unlike C-X ! you will be left in the new window. Now, the shell process is said to be attached to *shell-1*, and it is considered an *i-process* buffer.

13. Directory Handling

To save having to use absolute pathnames when you want to edit a nearby file JOVE allows you to move around the UNIX filesystem just as the c-shell does. These commands are:

<code>cd dir</code>	Change to the specified directory.
<code>pushd [dir]</code>	Like <i>cd</i> , but save the old directory on the directory stack. With no directory argument, simply exchange the top two directories on the stack and <i>cd</i> to the new top.
<code>popd</code>	Take the current directory off the stack and <i>cd</i> to the directory now at the top.
<code>dirs</code>	Display the contents of the directory stack.

The names and behavior of these commands were chosen to mimic those in the c-shell.

14. Editing C Programs

This section details the support provided by JOVE for working on C programs.

14.1. Indentation Commands

To save having to lay out C programs "by hand", JOVE has an idea of the correct indentation of a line, based on the surrounding context. When you are in C Mode, JOVE treats tabs specially — typing a tab at the beginning of a new line means "indent to the right place". Closing braces are also handled specially, and are indented to match the corresponding open brace.

14.2. Parenthesis and Brace Matching

To check that parentheses and braces match the way you think they do, turn on *Show Match* mode (ESC X *show-match-mode*). Then, whenever you type a close brace or parenthesis, the cursor moves momentarily to the matching opener, if it's currently visible. If it's not visible, JOVE displays the line containing the matching opener on the message line.

14.3. C Tags

Often when you are editing a C program, especially someone else's code, you see a function call and wonder what that function does. You then search for the function within the current file and if you're lucky find the definition, finally returning to the original spot when you are done. However, if are unlucky, the function turns out to be external (defined in another file) and you have to suspend the edit, *grep* for the function name in every .c that might contain it, and finally visit the appropriate file.

To avoid this diversion or the need to remember which function is defined in which file, Berkeley UNIX has a program called *ctags(1)*, which takes a set of source files and looks for function definitions, producing a file called *tags* as its output.

JOVE has a command called C-X T (*find-tag*) that prompts you for the name of a function (a *tag*), looks up the tag reference in the previously constructed tags file, then visits the file containing that tag in a new buffer, with point positioned at the definition of the function. There is another version of this command, namely *find-tag-at-point*, that uses the identifier at *point*.

So, when you've added new functions to a module, or moved some old ones around, run the *ctags* program to regenerate the *tags* file. JOVE looks in the file specified in the *tag-file* variable. The default

15.1.3. Lisp mode

This mode is analogous to *C Mode*, but performs the indentation needed to lay out Lisp programs properly. Note also the *grind-s-expr* command that prettyprints an *s-expression* and the *kill-mode-expression* command.

15.2. Minor Modes

In addition to the major modes, JOVE has a set of minor modes. These are as follows:

15.2.1. Auto Indent

In this mode, JOVE indents each line the same way as that above it. That is, the Return key in this mode acts as the Linefeed key ordinarily does.

15.2.2. Show Match

Move the cursor momentarily to the matching opening parenthesis when a closing parenthesis is typed.

15.2.3. Auto Fill

In *Auto Fill* mode, a newline is automatically inserted when the line length exceeds the right margin. This way, you can type a whole paper without having to use the Return key.

15.2.4. Over Write

In this mode, any text typed in will replace the previous contents. (The default is for new text to be inserted and "push" the old along.) This is useful for editing an already-formatted diagram in which you want to change some things without moving other things around on the screen.

15.2.5. Word Abbrev

In this mode, every word you type is compared to a list of word abbreviations; whenever you type an abbreviation, it is replaced by the text that it abbreviates. This can save typing if a particular word or phrase must be entered many times. The abbreviations and their expansions are held in a file that looks like:

```
abbrev:phrase
```

This file can be set up in your `~/.joverc` with the *read-word-abbrev-file* command. Then, whenever you are editing a buffer in *Word Abbrev* mode, JOVE checks for the abbreviations you've given. See also the commands *read-word-abbrev-file*, *write-word-abbrev-file*, *edit-word-abbrevs*, *define-global-word-abbrev*, *define-mode-word-abbrev*, and *bind-macro-to-word-abbrev*, and the variable *auto-case-abbrev*.

15.3. Variables

JOVE can be tailored to suit your needs by changing the values of variables. A JOVE variable can be given a value with the *set* command, and its value displayed with the *print* command.

The variables JOVE understands are listed along with the commands in the alphabetical list at the end of this document.

15.4. Key Re-binding

Many of the commands built into JOVE are not bound to specific keys. The command handler in JOVE is used to invoke these commands and is activated by the *execute-extended-command* command (ESC X). When the name of a command typed in is unambiguous, that command will be executed. Since it is very slow to have to type in the name of each command every time it is needed, JOVE makes it possible to *bind* commands to keys. When a command is *bound* to a key any future hits on that key will invoke that command. All the printing characters are initially bound to the command *self-insert*. Thus, typing any printing character causes it to be inserted into the text. Any of the

16. Alphabetical List of Commands and Variables

16.1. Prefix-1 (Escape)

This reads the next character and runs a command based on the character typed. If you wait for more than a second or so before typing the next character, the message "ESC" will be printed on the message line to remind you that JOVE is waiting for another character.

16.2. Prefix-2 (C-X)

This reads the next character and runs a command based on the character typed. If you wait for more than a second or so before typing another character, the message "C-X" will be printed on the message line to remind you that JOVE is waiting for another character.

16.3. Prefix-3 (Not Bound)

This reads the next character and runs a command based on the character typed. If you wait for more than a second or so before typing the next character, the character that invoked Prefix-3 will be printed on the message line to remind you that JOVE is waiting for another one.

16.4. allow-[^]S-and-[^]Q (variable)

This variable, when set, tells JOVE that your terminal does not need to use the characters C-S and C-Q for flow control, and that it is okay to bind things to them. This variable should be set depending upon what kind of terminal you have.

16.5. allow-bad-filenames (variable)

If set, this variable permits filenames to contain "bad" characters such as those from the set *%!"'[](). These files are harder to deal with, because the characters mean something to the shell. The default value is "off".

16.6. append-region (Not Bound)

This appends the region to a specified file. If the file does not already exist it is created.

16.7. apropos (Not Bound)

This types out all the commands, variables and macros with the specific keyword in their names. For each command and macro that contains the string, the key sequence that can be used to execute the command or macro is printed; with variables, the current value is printed. So, to find all the commands that are related to windows, you type "ESC X apropos window<Return>".

16.8. auto-case-abbrev (variable)

When this variable is on (the default), word abbreviations are adjusted for case automatically. For example, if "jove" were the abbreviation for "jonathan's own version of emacs", then typing "jove" would give you "jonathan's own version of emacs", typing "Jove" would give you "Jonathan's own version of emacs", and typing "JOVE" would give you "Jonathan's Own Version of Emacs". When this variable is "off", upper and lower case are distinguished when looking for the abbreviation, i.e., in the example above, "JOVE" and "Jove" would not be expanded unless they were defined separately.

16.9. auto-execute-command (Not Bound)

This tells JOVE to execute a command automatically when a file whose name matches a specified pattern is visited. The first argument is the command you want executed and the second is a regular expression pattern that specifies the files that apply. For example, if you want to be in show-match-mode when you edit C source files (that is, files that end with ".c" or ".h") you can type

```
ESC X auto-execute-command show-match-mode *.*[ch]$
```

16.20. beginning-of-line (C-A)

This moves point to the beginning of the current line.

16.21. beginning-of-window (ESC ,)

This moves point to the beginning of the current window. The sequence "ESC ," is the same as "ESC <" (beginning of file) except without the shift key on the "<", and can thus can easily be remembered.

16.22. bind-to-key (Not Bound)

This attaches a key to an internal JOVE command so that future hits on that key invoke that command. For example, to make "C-W" erase the previous word, you type "ESC X bind-to-key kill-previous-word C-W".

16.23. bind-macro-to-key (Not Bound)

This is like *bind-to-key* except you use it to attach keys to named macros.

16.24. bind-macro-to-word-abbrev (Not Bound)

This command allows you to bind a macro to a previously defined word abbreviation. Whenever you type the abbreviation, it will first be expanded as an abbreviation, and then the macro will be executed. Note that if the macro moves around, you should set the mark first (C-@) and then exchange the point and mark last (C-X C-X).

16.25. buffer-position (Not Bound)

This displays the current file name, current line number, total number of lines, percentage of the way through the file, and the position of the cursor in the current line.

16.26. c-mode (Not Bound)

This turns on C mode in the currently selected buffer. This is one of currently four possible major modes: Fundamental, Text, C, Lisp. When in C or Lisp mode, Tab, ")", and ")" behave a little differently from usual: They are indented to the "right" place for C (or Lisp) programs. In JOVE, the "right" place is simply the way the author likes it (but I've got good taste).

16.27. case-character-capitalize (Not Bound)

This capitalizes the character after point, i.e., the character under the cursor. If a negative argument is supplied that many characters *before* point are upper cased.

16.28. case-ignore-search (variable)

This variable, when set, tells JOVE to treat upper and lower case as the same when searching. Thus "jove" and "JOVE" would match, and "JoVe" would match either. The default value of this variable is "off".

16.29. case-region-lower (Not Bound)

This changes all the upper case letters in the region to their lower case equivalent.

16.30. case-region-upper (Not Bound)

This changes all the lower case letters in the region to their upper case equivalent.

16.31. case-word-capitalize (ESC C)

This capitalizes the current word by making the current letter upper case and making the rest of the word lower case. Point is moved to the end of the word. If point is not positioned on a word it is first moved forward to the beginning of the next word. If a negative argument is supplied that many words *before* point are capitalized. This is useful for correcting the word just typed without having to

16.41. current-error (Not Bound)

This moves to the current error in the list of parsed errors. See the *next-error* and *previous-error* commands for more detailed information.

16.42. date (Not Bound)

This prints the date on the message line.

16.43. define-mode-word-abbrev (Not Bound)

This defines a mode-specific abbreviation.

16.44. define-global-word-abbrev (Not Bound)

This defines a global abbreviation.

16.45. delete-blank-lines (C-X C-O)

This deletes all the blank lines around point. This is useful when you previously opened many lines with "C-O" and now wish to delete the unused ones.

16.46. delete-buffer (C-X K)

This deletes a buffer and frees up all the memory associated with it. Be careful! Once a buffer has been deleted it is gone forever. JOVE will ask you to confirm if you try to delete a buffer that needs saving. This command is useful for when JOVE runs out of space to store new buffers.

16.47. delete-macro (Not Bound)

This deletes a macro from the list of named macros. It is an error to delete the keyboard-macro. Once the macro is deleted it is gone forever. If you are about to save macros to a file and decide you don't want to save a particular one, delete it.

16.48. delete-next-character (C-D)

This deletes the character that's just after point (that is, the character under the cursor). If point is at the end of a line, the line separator is deleted and the next line is joined with the current one.

16.49. delete-other-windows (C-X 1)

This deletes all the other windows except the current one. This can be thought of as going back into One Window mode.

16.50. delete-previous-character (Rubout)

This deletes the character that's just before point (that is, the character before the cursor). If point is at the beginning of the line, the line separator is deleted and that line is joined with the previous one.

16.51. delete-white-space (ESC \)

This deletes all the Tabs and Spaces around point.

16.52. delete-current-window (C-X D)

This deletes the current window and moves point into one of the remaining ones. It is an error to try to delete the only remaining window.

16.53. describe-bindings (Not Bound)

This types out a list containing each bound key and the command that gets invoked every time that key is typed. To make a wall chart of JOVE commands, set *send-typeout-to-buffer* to "on" and JOVE will store the key bindings in a buffer which you can save to a file and then print.

16.65. digit-8 (Not Bound)

This pretends you typed "ESC 8". This is useful for terminals that have keypads that send special sequences for numbers typed on the keypad as opposed to numbers typed from the keyboard. This can save having type "ESC" when you want to specify an argument.

16.66. digit-9 (Not Bound)

This pretends you typed "ESC 9". This is useful for terminals that have keypads that send special sequences for numbers typed on the keypad as opposed to numbers typed from the keyboard. This can save having type "ESC" when you want to specify an argument.

16.67. digit-0 (Not Bound)

This pretends you typed "ESC 0". This is useful for terminals that have keypads that send special sequences for numbers typed on the keypad as opposed to numbers typed from the keyboard. This can save having type "ESC" when you want to specify an argument.

16.68. dirs (Not Bound)

This prints out the directory stack. See the "cd", "pushd", "popd" commands for more info.

16.69. disable-biff (variable)

When this is set, JOVE disables biff when you're editing and enables it again when you get out of JOVE, or when you pause to the parent shell or push to a new shell. (This means arrival of new mail will not be immediately apparent but will not cause indiscriminate writing on the display). The default is "off".

16.70. dstop-process (Not Bound)

Send the "dsusp" character to the current process. This is the character that suspends a process on the next read from the terminal. Most people have it set to C-Y. This only works if you have the interactive process feature, and if you are in a buffer bound to a process.

16.71. edit-word-abbrevs (Not Bound)

This creates a buffer with a list of each abbreviation and the phrase it expands into, and enters a recursive edit to let you change the abbreviations or add some more. The format of this list is "abbreviation:phrase" so if you add some more you should follow that format. It's probably simplest just to copy some already existing abbreviations and edit them. When you are done you type "C-X C-C" to exit the recursive edit.

16.72. end-of-file (ESC >)

This moves point forward to the end of the buffer. This sometimes prints the "Point Pushed" message. If the end of the buffer isn't on the screen JOVE will set the mark so you can go back to where you were if you want.

16.73. end-of-line (C-E)

This moves point to the end of the current line. If the line is too long to fit on the screen JOVE will scroll the line to the left to make the end of the line visible. The line will slide back to its normal position when you move backward past the leftmost visible character or when you move off the line altogether.

16.74. end-of-window (ESC .)

This moves point to the last character in the window.

it will add one automatically.

16.85. fill-comment (Not Bound)

This command fills in your C comments to make them pretty and readable. This filling is done according to the variable *comment-format*.

```
/*  
* the default format makes comments like this.  
*/
```

This can be changed by changing the format variable. Other languages may be supported by changing the format variable appropriately. The formatter looks backwards from dot for an open comment symbol. If found, all indentation is done relative the position of the first character of the open symbol. If there is a matching close symbol, the entire comment is formatted. If not, the region between dot and the open symbol is reformatted.

16.86. fill-paragraph (ESC J)

This rearranges words between lines so that all the lines in the current paragraph extend as close to the right margin as possible, ensuring that none of the lines will be greater than the right margin. The default value for *right-margin* is 78, but can be changed with the *set* and *right-margin-here* commands. JOVE has a complicated algorithm for determining the beginning and end of the paragraph. In the normal case JOVE will give all the lines the same indent as they currently have, but if you wish to force a new indent you can supply a numeric argument to *fill-paragraph* (e.g., by typing C-U ESC J) and JOVE will indent each line to the column specified by the *left-margin* variable. See also the *left-margin* variable and *left-margin-here* command.

16.87. fill-region (Not Bound)

This is like *fill-paragraph*, except it operates on a region instead of just a paragraph.

16.88. filter-region (Not Bound)

This sends the text in the region to a UNIX command, and replaces the region with the output from that command. For example, if you are lazy and don't like to take the time to write properly indented C code, you can put the region around your C file and *filter-region* it through *cb*, the UNIX C beautifier. If you have a file that contains a bunch of lines that need to be sorted you can do that from inside JOVE too, by filtering the region through the *sort* UNIX command. Before output from the command replaces the region JOVE stores the old text in the kill ring, so if you are unhappy with the results you can easily get back the old text with "C-Y".

16.89. find-file (C-X C-F)

This visits a file into its own buffer and then selects that buffer. If you've already visited this file in another buffer, that buffer is selected. If the file doesn't yet exist, JOVE will print "(New file)" so that you know.

16.90. find-tag (C-X T)

This finds the file that contains the specified tag. JOVE looks up tags by default in the "tags" file in the current directory. You can change the default tag name by setting the *tag-file* variable to another name. If you specify a numeric argument to this command, you will be prompted for a tag file. This is a good way to specify another tag file without changing the default. If the tag cannot be found the error is reported and point stays where it is.

16.91. find-tag-at-point (Not Bound)

This finds the file that contains the tag that point is currently on. See *find-tag*.

16.104. i-search-forward (Not Bound)

Incremental search. Like search-forward except that instead of prompting for a string and searching for that string all at once, it accepts the string one character at a time. After each character you type as part of the search string, it searches for the entire string so far. When you like what it found, type the Return key to finish the search. You can take back a character with Rubout and the search will back up to the position before that character was typed. C-G aborts the search.

16.105. i-search-reverse (Not Bound)

Incremental search. Like search-reverse except that instead of prompting for a string and searching for that string all at once, it accepts the string one character at a time. After each character you type as part of the search string, it searches for the entire string so far. When you like what it found, type the Return key to finish the search. You can take back a character with Rubout and the search will back up to the position before that character was typed. C-G aborts the search.

16.106. insert-file (C-X C-I)

This inserts a specified file into the current buffer at point. Point is positioned at the beginning of the inserted file.

16.107. internal-tabstop (variable)

The number of spaces JOVE should print when it displays a tab character. The default value is 8.

16.108. interrupt-process (Not Bound)

This sends the interrupt character (usually C-C) to the interactive process in the current buffer. This is only for versions of JOVE that have the interactive processes feature. This only works when you are inside a buffer that's attached to a process.

16.109. i-shell (Not Bound)

This starts up an interactive shell in a window. JOVE uses "shell-1" as the name of the buffer in which the interacting takes place. See the manual for information on how to use interactive processes.

16.110. i-shell-command (Not Bound)

This is like *shell-command* except it lets you continue with your editing while the command is running. This is really useful for long running commands with sporadic output. See the manual for information on how to use interactive processes.

16.111. kill-next-word (ESC D)

This kills the text from point to the end of the current or next word.

16.112. kill-previous-word (ESC Rubout)

This kills the text from point to the beginning of the current or previous word.

16.113. kill-process (Not Bound)

This command prompts for a buffer name or buffer number (just as select-buffer does) and then sends the process in that buffer a kill signal (9).

16.114. kill-region (C-W)

This deletes the text in the region and saves it on the kill ring. Commands that delete text but save it on the kill ring all have the word "kill" in their names. Type "C-Y" to yank back the most recent kill.

column is the name of the file that's attached to the buffer. In this case, both Minibuf and commands.doc have been changed but not yet saved. In fact Minibuf won't be saved since it's an internal JOVE buffer that I don't even care about.

16.124. list-processes (Not Bound)

This makes a list somewhat like "list-buffers" does, except its list consists of the current interactive processes. Right now the list looks like this:

Buffer	Status	Command name
-----	-----	-----
shell-1	Running	i-shell
fgrep	Done	fgrep -n Buffer *.c

The first column has the name of the buffer to which the process is attached. The second has the status of the process; if a process has exited normally the status is "Done" as in fgrep; if the process exited with an error the status is "Exit N" where N is the value of the exit code; if the process was killed by some signal the status is the name of the signal that was used; otherwise the process is running. The last column is the name of the command that is being run.

16.125. mailbox (variable)

Set this to the full pathname of your mailbox. JOVE will look here to decide whether or not you have any unread mail. This defaults to /usr/spool/mail/\$USER, where \$USER is set to your login name.

16.126. mail-check-frequency (variable)

This is how often (in seconds) JOVE should check your mailbox for incoming mail. See also the *mailbox* and *disable-biff* variables.

16.127. make-backup-files (variable)

If this variable is set, then whenever JOVE writes out a file, it will move the previous version of the file (if there was one) to "#filename". This is often convenient if you save a file by accident. The default value of this variable is "off". *Note:* this is an optional part of JOVE, and your guru may not have it enabled, so it may not work.

16.128. make-buffer-unmodified (ESC ~)

This makes JOVE think the selected buffer hasn't been changed even if it has. Use this when you accidentally change the buffer but don't want it considered changed. Watch the mode line to see the * disappear when you use this command.

16.129. make-macro-interactive (Not Bound)

This command is meaningful only while you are defining a keyboard macro. Ordinarily, when a command in a macro definition requires a trailing text argument (file name, search string, etc.), the argument you supply becomes part of the macro definition. If you want to be able to supply a different argument each time the macro is used, then while you are defining it, you should give the make-macro-interactive command just before typing the argument which will be used during the definition process. *Note:* you must bind this command to a key in order to use it; you can't say ESC X make-macro-interactive.

16.130. mark-threshold (variable)

This variable contains the number of lines point may move by before the mark is set. If, in a search or something, point moves by more than this many lines, the mark is set so that you may return easily. The default value of this variable is 22 (one screenful, on most terminals).

16.136. name-keyboard-macro (Not Bound)

This copies the keyboard macro and gives it a name freeing up the keyboard macro so you can define some more. Keyboard macros with their own names can be bound to keys just like built in commands can. See the *read-macros-file-file* and *write-macros-to-file* commands.

16.137. newline (Return)

This divides the current line at point moving all the text to the right of point down onto the newly created line. Point moves down to the beginning of the new line.

16.138. newline-and-backup (C-O)

This divides the current line at point moving all the text to the right of point down onto the newly created line. The difference between this and "newline" is that point does not move down to the beginning of the new line.

16.139. newline-and-indent (LineFeed)

This behaves the same as Return does when in Auto Indent mode. This makes Auto Indent mode obsolete but it remains in the name of backward compatibility.

16.140. next-error (C-X C-N)

This moves to the next error in the list of errors that were parsed with *parse-errors* or *parse-special-errors*. In one window the list of errors is shown with the current one always at the top. In another window is the file that contains the error. Point is positioned in this window on the line where the error occurred.

16.141. next-line (C-N)

This moves down to the next line.

16.142. next-page (C-V)

This displays the next page of the buffer by taking the bottom line of the window and redrawing the window with it at the top. If there isn't another page in the buffer JOVE rings the bell. If a numeric argument is supplied the screen is scrolled up that many lines; if the argument is negative the screen is scrolled down.

16.143. next-window (C-X N)

This moves into the next window. Windows live in a circular list so when you're in the bottom window and you try to move to the next one you are moved to the top window. It is an error to use this command with only one window.

16.144. number-lines-in-window (Not Bound)

This displays the line numbers for each line in the buffer being displayed. The number isn't actually part of the text; it's just printed before the actual buffer line is. To turn this off you run the command again; it toggles.

16.145. over-write-mode (Not Bound)

This turns Over Write mode on (or off if it's currently on) in the selected buffer. When on, this mode changes the way the self-inserting characters work. Instead of inserting themselves and pushing the rest of the line over to the right, they replace or over-write the existing character. Also, Rubout replaces the character before point with a space instead of deleting it. When Over Write mode is on "OvrWt" is displayed on the mode line.

16.157. previous-page (ESC V)

This displays the previous page of the current buffer by taking the top line and redrawing the window with it at the bottom. If a numeric argument is supplied the screen is scrolled down that many lines; if the argument is negative the screen is scrolled up.

16.158. previous-window (C-X P and C-X O)

This moves into the next window. Windows live in a circular list so when you're in the top window and you try to move to the previous one you are moved to the bottom window. It is an error to use this command with only one window.

16.159. print (Not Bound)

This prints the value of a JOVE variable.

16.160. print-message (Not Bound)

This command prompts for a message, and then prints it on the bottom line where JOVE messages are printed.

16.161. process-bind-to-key (Not Bound)

This command is identical to `bind-to-key`, except that it only affects your bindings when you are in a buffer attached to a process. When you enter the process buffer, any keys bound with this command will automatically take their new values. When you switch to a non-process buffer, the old bindings for those keys will be restored. For example, you might want to execute

```
process-bind-to-key stop-process ^Z
process-bind-to-key interrupt-process ^C
```

Then, when you start up an interactive process and switch into that buffer, C-Z will execute `stop-process` and C-C will execute `interrupt-process`. When you switch back to a non-process buffer, C-Z will go back to executing `scroll-up` (or whatever you have it bound to).

16.162. process-newline (Return)

This this only gets executed when in a buffer that is attached to an interactive-process. JOVE does two different things depending on where you are when you hit Return. When you're at the end of the I-Process buffer this does what Return normally does, except it also makes the line available to the process. When point is positioned at some other position that line is copied to the end of the buffer (with the prompt stripped) and point is moved there with it, so you can then edit that line before sending it to the process. This command *must* be bound to the key you usually use to enter shell commands (Return), or else you won't be able to enter any.

16.163. process-prompt (variable)

What a prompt looks like from the i-shell and i-shell-command processes. The default is "% ", the default C-shell prompt. This is actually a regular expression search string. So you can set it to be more than one thing at once using the `\|` operator. For instance, for LISP hackers, the prompt can be "% \|-> \|<[0-9]>:".

16.164. push-shell (Not Bound)

This spawns a child shell and relinquishes control to it. This works on any version of UNIX, but this isn't as good as *pause-jove* because it takes time to start up the new shell and you get a brand new environment every time. To return to JOVE you type "C-D".

16.172. read-macros-from-file (Not Bound)

This reads the specified file that contains a bunch of macro definitions, and defines all the macros that were currently defined when the file was created. See *write-macros-to-file* to see how to save macros.

16.173. redraw-display (C-L)

This centers the line containing point in the window. If that line is already in the middle the window is first cleared and then redrawn. If a numeric argument is supplied, the line is positioned at that offset from the top of the window. For example, "ESC 0 C-L" positions the line containing point at the top of the window.

16.174. recursive-edit (Not Bound)

This enters a recursive editing level. This isn't really very useful. I don't know why it's available for public use. I think I'll delete it some day.

16.175. rename-buffer (Not Bound)

This lets you rename the current buffer.

16.176. replace-in-region (Not Bound)

This is the same as *replace-string* except that it is restricted to occurrences between Point and Mark.

16.177. replace-string (ESC R)

This replaces all occurrences of a specified string with a specified replacement string. This is just like *query-replace-string* except it replaces without asking.

16.178. right-margin (variable)

Where the right margin is for *Auto Fill* mode and the *justify-paragraph* and *justify-region* commands. The default is 78.

16.179. right-margin-here (Not Bound)

This sets the *right-margin* variable to the current position of point. This is an easy way to say, "Make the right margin begin here," without having to count the number of spaces over it actually is.

16.180. save-file (C-X C-S)

This saves the current buffer to the associated file. This makes your changes permanent so you should be sure you really want to. If the buffer has not been modified *save-file* refuses to do the save. If you really do want to write the file you can use "C-X C-W" which executes *write-file*.

16.181. scroll-down (ESC Z)

This scrolls the screen one line down. If the line containing point moves past the bottom of the window point is moved up to the center of the window. If a numeric argument is supplied that many lines are scrolled; if the argument is negative the screen is scrolled up instead.

16.182. scroll-step (variable)

How many lines should be scrolled if the *previous-line* or *next-line* commands move you off the top or bottom of the screen. You may wish to decrease this variable if you are on a slow terminal.

16.183. scroll-up (C-Z)

This scrolls the screen one line up. If the line containing point moves past the top of the window point is moved down to the center of the window. If a numeric argument is supplied that many lines are scrolled; if the argument is negative the screen is scrolled down instead.

holding a file, not some output from a previous command, JOVE prints an error message and refuses to execute the command. If you really want to execute the command you should delete that buffer (saving it first, if you like) or use *shell-command-to-buffer*, and try again.

16.194. *shell-command-to-buffer* (Not Bound)

This is just like *shell-command* except it lets you specify the buffer to use instead of JOVE.

16.195. *shell-flags* (variable)

This defines the flags that are passed to shell commands. The default is "-c". See the *shell* variable to change the default shell.

16.196. *show-match-mode* (Not Bound)

This turns on Show Match mode (or off if it's currently on) in the selected buffer. This changes ")" and ")" so that when they are typed they are inserted as usual, and then the cursor flashes back to the matching "(" or "(" (depending on what was typed) for about half a second, and then goes back to just after the ")" or ")" that invoked the command. This is useful for typing in complicated expressions in a program. You can change how long the cursor sits on the matching paren by setting the "paren-flash-delay" variable in tenths of a second. If the matching "(" or "(" isn't visible nothing happens.

16.197. *shrink-window* (Not Bound)

This makes the current window one line shorter, if possible. Windows must be at least 2 lines high, one for the text and the other for the mode line.

16.198. *source* (Not Bound)

This reads a bunch of JOVE commands from a file. The format of the file is the same as that in your initialization file (your ".joverc") in your main directory. There should be one command per line and it should be as though you typed "ESC X" while in JOVE. For example, here's part of my initialization file:

```
bind-to-key i-search-reverse ^R
bind-to-key i-search-forward ^S
bind-to-key pause-jove ^[S
```

What they do is make "C-R" call the *i-search-reverse* command and "C-S" call *i-search-forward* and "ESC S" call *pause-jove*.

16.199. *spell-buffer* (Not Bound)

This runs the current buffer through the UNIX *spell* program and places the output in buffer "Spell". Then JOVE lets you edit the list of words, expecting you to delete the ones that you don't care about, i.e., the ones you know are spelled correctly. Then the *parse-spelling-errors-in-buffer* command comes along and finds all the misspelled words and sets things up so the error commands work.

16.200. *split-current-window* (C-X 2)

This splits the current window into two equal parts (providing the resulting windows would be big enough) and displays the selected buffer in both windows. Use "C-X 1" to go back to 1 window mode.

16.201. *start-remembering* (C-X)

This starts remembering your key strokes in the Keyboard macro. To stop remembering you type "C-X)". Because of a bug in JOVE you can't stop remembering by typing "ESC X stop-remembering"; *stop-remembering* must be bound to "C-X)" in order to make things work correctly. To execute the remembered key strokes you type "C-X E" which runs the *execute-keyboard-macro* command. Sometimes you may want a macro to accept different input each time it runs. To see how to do this, see

16.213. use-i/d-char (variable)

If your terminal has insert/delete character capability you can tell JOVE not to use it by setting this to "off". In my opinion it is only worth using insert/delete character at low baud rates. **WARNING:** if you set this to "on" when your terminal doesn't have insert/delete character capability, you will get weird (perhaps fatal) results.

16.214. version (Not Bound)

Displays the version number of this JOVE.

16.215. visible-bell (variable)

Use the terminal's visible bell instead of beeping. This is set automatically if your terminal has the capability.

16.216. visible-spaces-in-window (Not Bound)

This displays an underscore character instead of each space in the window and displays a greater-than followed by spaces for each tab in the window. The actual text in the buffer is not changed; only the screen display is affected. To turn this off you run the command again; it toggles.

16.217. visit-file (C-X C-V)

This reads a specified file into the current buffer replacing the old text. If the buffer needs saving JOVE will offer to save it for you. Sometimes you use this to start over, say if you make lots of changes and then change your mind. If that's the case you don't want JOVE to save your buffer and you answer "NO" to the question.

16.218. window-find (C-X 4)

This lets you select another buffer in another window three different ways. This waits for another character which can be one of the following:

T	Finds a tag in the other window.
F	Finds a file in the other window.
B	Selects a buffer in the other window.

This is just a convenient short hand for "C-X 2" (or "C-X O" if there are already two windows) followed by the appropriate sequence for invoking each command. With this, though, there isn't the extra overhead of having to redisplay. In addition, you don't have to decide whether to type "C-X 2" or "C-X O" since "C-X 4" does the right thing.

16.219. word-abbrev-mode (Not Bound)

This turns on Word Abbrev mode (or off if it's currently on) in the selected buffer. Word Abbrev mode lets you specify a word (an abbreviation) and a phrase with which JOVE should substitute the abbreviation. You can use this to define words to expand into long phrases, e.g., "jove" can expand into "Jonathan's Own Version of Emacs"; another common use is defining words that you often misspell in the same way, e.g., "thier" => "their" or "teh" => "the". See the information on the *auto-case-abbrev* variable. There are two kinds of abbreviations: mode specific and global. If you define a Mode specific abbreviation in C mode, it will expand only in buffers that are in C mode. This is so you can have the same abbreviation expand to different things depending on your context. Global abbreviations expand regardless of the major mode of the buffer. The way it works is this: JOVE looks first in the mode specific table, and then in the global table. Whichever it finds it in first is the one that's used in the expansion. If it doesn't find the word it is left untouched. JOVE tries to expand words as they are typed, when you type a punctuation character or Space or Return. If you are in Auto Fill mode the expansion will be filled as if you typed it yourself.

SED — A Non-interactive Text Editor

Lee E. McMahon

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

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 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.)

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

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.



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:

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.





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* function copies the contents of the pattern space into a hold area (destroying the previous contents of the hold area).

(2)H -- Hold pattern space

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

(2)g -- get contents of hold area

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

(2)G -- Get contents of hold area

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

(2)x -- exchange

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

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 address 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.

Awk — A Pattern Scanning and Processing Language (Second Edition)

Alfred V. Aho

Brian W. Kernighan

Peter J. Weinberger

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Awk is a programming language whose basic operation is to search a set of files for patterns, and to perform specified actions upon lines or fields of lines which contain instances of those patterns. *Awk* makes certain data selection and transformation operations easy to express; for example, the *awk* program

length > 72

prints all input lines whose length exceeds 72 characters; the program

NF % 2 == 0

prints all lines with an even number of fields; and the program

(\$1 = log(\$1); print)

replaces the first field of each line by its logarithm.

Awk patterns may include arbitrary boolean combinations of regular expressions and of relational operators on strings, numbers, fields, variables, and array elements. Actions may include the same pattern-matching constructions as in patterns, as well as arithmetic and string expressions and assignments, **if-else**, **while**, **for** statements, and multiple output streams.

This report contains a user's guide, a discussion of the design and implementation of *awk*, and some timing statistics.

1. Introduction

Awk is a programming language designed to make many common information retrieval and text manipulation tasks easy to state and to perform.

The basic operation of *awk* is to scan a set of input lines in order, searching for lines which match any of a set of patterns which the user has specified. For each pattern, an action can be specified; this action will be performed on each line that matches the pattern.

Readers familiar with the UNIX† program

*grep*¹ will recognize the approach, although in *awk* the patterns may be more general than in *grep*, and the actions allowed are more involved than merely printing the matching line. For example, the *awk* program

(print \$3, \$2)

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

\$2 ~ /A|B|C/

prints all input lines with an A, B, or C in the

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

Similarly, output can be piped into another process (on UNIX only); for instance,

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

mails the output to **bwk**.

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

Awk also provides the **printf** statement for output formatting:

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

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

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

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

2. Patterns

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

2.1. BEGIN and END

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

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

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

Or the input lines may be counted by

```
END { print NR }
```

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

2.2. Regular Expressions

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

```
/smith/
```

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

```
blacksmithing
```

Awk regular expressions include the regular expression forms found in the UNIX text editor *ed*¹ and *grep* (without back-referencing). In addition, *awk* allows parentheses for grouping, | for alternatives, + for "one or more", and ? for "zero or one", all as in *lex*. Character classes may be abbreviated: **[a-zA-Z0-9]** is the set of all letters and digits. As an example, the *awk* program

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

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

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

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

which matches any string of characters enclosed in slashes.

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

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

prints all lines where the first field matches "john" or "John." Notice that this will also match "Johnson", "St. Johnsberry", and so on. To restrict it to exactly **[jJ]ohn**, use

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

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

2.3. Relational Expressions

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

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

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

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

prints lines with an even number of fields.

Arithmetic is done internally in floating point. The arithmetic operators are +, -, *, /, and % (mod). The C increment ++ and decrement -- operators are also available, and so are the assignment operators +=, -=, *=, /=, and %= . These operators may all be used in expressions.

3.3. Field Variables

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

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

or accumulate two fields into a third, like this:

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

or assign a string to a field:

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

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

Field references may be numerical expressions, as in

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

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

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

fields are treated as strings.

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

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

splits the the string *s* into *array*[1], ..., *array*[*n*]. The number of elements found is returned. If the *sep* argument is provided, it is used as the field separator; otherwise FS is used as the separator.

3.4. String Concatenation

Strings may be concatenated. For example

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

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

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

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

3.5. Arrays

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

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

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

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

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

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

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

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

3.6. Flow-of-Control Statements

Awk provides the basic flow-of-control statements *if-else*, *while*, *for*, and statement grouping with braces, as in C. We showed the *if* statement in section 3.3 without describing it. The condition in parentheses is evaluated; if it is true, the statement following the *if* is done. The *else* part is optional.

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

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

The *for* statement is also exactly that of C:

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

does the same job as the *while* statement above.

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

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

does *statement* with *i* set in turn to each element of *array*. The elements are accessed in an apparently



“jdmr”, respectively.

7. print each line prefixed by “line-number :”.
8. sum the fourth column of a table.

The program *wc* merely counts words, lines and characters in its input; we have already mentioned the others. In all cases the input was a file containing 10,000 lines as created by the command *ls -l*; each line has the form

```
-rw-rw-rw- 1 ava 123 Oct 15 17:05 xxx
```

The total length of this input is 452,960 characters. Times for *lex* do not include compile or load.

As might be expected, *awk* is not as fast as the specialized tools *wc*, *sed*, or the programs in the *grep* family, but is faster than the more general tool *lex*. In all cases, the tasks were about as easy to express as *awk* programs as programs in these other languages; tasks involving fields were considerably easier to express as *awk* programs. Some of the test programs are shown in *awk*, *sed* and *lex*.

References

1. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual*, Bell Laboratories, May 1975. Sixth Edition
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
3. M. E. Lesk, “Lex — A Lexical Analyzer Generator,” Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, October 1975. Reprinted as PS1:16 in *UNIX Programmer's Manual*, Usenix Association, (1986).
4. S. C. Johnson, “Yacc — Yet Another Compiler-Compiler,” Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975. Reprinted as PS1:15 in *UNIX Programmer's Manual*, Usenix Association, (1986).



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

M. E. Lesk

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

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

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

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

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

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

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

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

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

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

To omit some of the standard headings (e.g. no abstract, or no author's institution) just omit the



```
.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



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

whereas

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

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

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

.BX word

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

Longer pieces of text may be boxed by enclosing them with .B1 and .B2:

```
.B1
text...
.B2
```

as has been done here.

Keeping blocks together. If

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



Appendix A List of Commands

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

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	XX



A Revised Version of -ms

Bill Tuthill

Computing Services
University of California
Berkeley, CA 94720

The -ms macros have been slightly revised and rearranged for the Berkeley Unix distribution. Because of the rearrangement, the new macros can be read by the computer in about half the time required by the previous version of -ms. This means that output will begin to appear between ten seconds and several minutes more quickly, depending on the system load. On long files, however, the savings in total time are not substantial. The old version of -ms is still available as -mos.

Several bugs in -ms have been fixed, including a bad problem with the .1C macro, minor difficulties with boxed text, a break induced by .EQ before initialization, the failure to set tab stops in displays, and several bothersome errors in the refer macros. Macros used only at Bell Laboratories have been removed. There are a few extensions to previous -ms macros, and a number of new macros, but all the documented -ms macros still work exactly as they did before, and have the same names as before. Output produced with -ms should look like output produced with -mos.

One important new feature is automatically numbered footnotes. Footnote numbers are printed by means of a pre-defined string (**), which you invoke separately from .FS and .FE. Each time it is used, this string increases the footnote number by one, whether or not you use .FS and .FE in your text. Footnote numbers will be superscripted on the phototypesetter and on daisy-wheel terminals, but on low-resolution devices (such as the lpr and a crt), they will be bracketed. If you use ** to indicate numbered footnotes, then the .FS macro will automatically include the footnote number at the bottom of the page. This footnote, for example, was produced as follows:¹

```
This footnote, for example, was produced as follows:\**
.FS
...
.FE
```

If you are using ** to number footnotes, but want a particular footnote to be marked with an asterisk or a dagger, then give that mark as the first argument to .FS: †

```
then give that mark as the first argument to .FS: \{dg
.FS \{dg
...
.FE
```

Footnote numbering will be temporarily suspended, because the ** string is not used. Instead of a dagger, you could use an asterisk * or double dagger ‡, represented as \{dd.

Another new feature is a macro for printing theses according to Berkeley standards. This macro is called .TM, which stands for thesis mode. (It is much like the .th macro in -me.) It will put page numbers in the upper right-hand corner; number the first page; suppress the date; and double-space everything except quotes, displays, and keeps. Use it at the top of each file making up your thesis.

¹ If you never use the "**" string, no footnote numbers will appear anywhere in the text, including down here. The output footnotes will look exactly like footnotes produced with -mos.

† In the footnote, the dagger will appear where the footnote number would otherwise appear, as on the left.



There are now a large number of optional foreign accent marks defined by the -ms macros. All the accent marks available in -mos are present, and they all work just as they always did. However, there are better definitions available by placing .AM at the beginning of your document. Unlike the -mos accent marks, the accent strings should come *after* the letter being accented. Here is a list of the diacritical marks, with examples of what they look like.

name of accent	input	output
acute accent	e\ <i>*</i> '	é
grave accent	e\ <i>*</i> `	è
circumflex	o\ <i>*</i> ^	ô
cedilla	c\ <i>*</i> ,	ç
tilde	n\ <i>*</i> ~	ñ
question	*?	¿
exclamation	*!	¡
umlaut	u\ <i>*</i> :	ü
digraph s	*8	ß
háček	c\ <i>*</i> v	č
macron	a\ <i>*</i> _	ā
underdot	s\ <i>*</i> .	š
o-slash	o\ <i>*</i> /	ø
angstrom	a\ <i>*</i> o	ång
yogh	kni\ <i>*</i> 3t	kniȝt
Thorn	*(Th	Þ
thorn	*(th	þ
Eth	*(D-	Ð
eth	*(d-	ð
hooked o	*q	ø
ae ligature	*(ae	æ
AE ligature	*(Ae	Æ
oe ligature	*(oe	œ
OE ligature	*(Oe	Œ

If you want to use these new diacritical marks, don't forget the .AM at the top of your file. Without it, some will not print at all, and others will be placed on the wrong letter.

It is also possible to produce custom headers and footers that are different on even and odd pages. The .OH and .EH macros define odd and even headers, while .OF and .EF define odd and even footers. Arguments to these four macros are specified as with .tl. This document was produced with:

```
.OH \fiThe -mx Macros Page %\fP'
.EH \fiPage %The -mx Macros\fP'
```

Note that it would be a error to have an apostrophe in the header text; if you need one, you will have to use a different delimiter around the left, center, and right portions of the title. You can use any character as a delimiter, provided it doesn't appear elsewhere in the argument to .OH, .EH, .OF, or EF.

The -ms macros work in conjunction with the **tbl**, **eqn**, and **refer** preprocessors. Macros to deal with these items are read in only as needed, as are the thesis macros (.TM), the special accent mark definitions (.AM), table of contents macros (.XS and .XE), and macros to format the optional cover page. The code for the -ms package lives in /usr/lib/tmac/tmac.s, and sourced files reside in the directory /usr/ucb/lib/ms.

Writing Papers with NROFF using -me

Eric P. Allman*

Project INGRES
Electronics Research Laboratory
University of California, Berkeley
Berkeley, California 94720

This document describes the text processing facilities available on the UNIX† operating system via NROFF† and the -me macro package. It is assumed that the reader already is generally familiar with the UNIX operating system and a text editor such as ex. This is intended to be a casual introduction, and as such not all material is covered. In particular, many variations and additional features of the -me macro package are not explained. For a complete discussion of this and other issues, see *The -me Reference Manual* and *The NROFF/TROFF Reference Manual*.

NROFF, a computer program that runs on the UNIX operating system, reads an input file prepared by the user and outputs a formatted paper suitable for publication or framing. The input consists of *text*, or words to be printed, and *requests*, which give instructions to the NROFF program telling how to format the printed copy.

Section 1 describes the basics of text processing. Section 2 describes the basic requests. Section 3 introduces displays. Annotations, such as footnotes, are handled in section 4. The more complex requests which are not discussed in section 2 are covered in section 5. Finally, section 6 discusses things you will need to know if you want to typeset documents. If you are a novice, you probably won't want to read beyond section 4 until you have tried some of the basic features out.

When you have your raw text ready, call the NROFF formatter by typing as a request to the UNIX shell:

```
nroff -me -Ttype files
```

where *type* describes the type of terminal you are outputting to. Common values are *dtc* for a DTC 300s (daisy-wheel type) printer and *lpr* for the line printer. If the -T flag is omitted, a "lowest common denominator" terminal is assumed; this is good for previewing output on most terminals. A complete description of options to the NROFF command can be found in *The NROFF/TROFF Reference Manual*.

The word *argument* is used in this manual to mean a word or number which appears on the same line as a request which modifies the meaning of that request. For example, the request

```
.sp
```

spaces one line, but

*Author's current address: Britton Lee, Inc., 1919 Addison Suite 105, Berkeley, California 94704.

†UNIX is a trademark of AT&T Bell Laboratories



Now is the time for all good men to come to the aid of their party.
Four score and seven years ago,...

Notice that the sentences of the paragraphs *must not* begin with a space, since blank lines and lines beginning with spaces cause a break. For example, if I had typed:

```
.pp
Now is the time for all good men
  to come to the aid of their party.
Four score and seven years ago,...
```

The output would be:

```
Now is the time for all good men
  to come to the aid of their party. Four score and seven years ago,...
```

A new line begins after the word "men" because the second line began with a space character.

There are many fancier types of paragraphs, which will be described later.

2.2. Headers and Footers

Arbitrary headers and footers can be put at the top and bottom of every page. Two requests of the form `.he title` and `.fo title` define the titles to put at the head and the foot of every page, respectively. The titles are called *three-part* titles, that is, there is a left-justified part, a centered part, and a right-justified part. To separate these three parts the first character of *title* (whatever it may be) is used as a delimiter. Any character may be used, but backslash and double quote marks should be avoided. The percent sign is replaced by the current page number whenever found in the title. For example, the input:

```
.he "%~
.fo 'Jane Jones'~My Book'
```

results in the page number centered at the top of each page, "Jane Jones" in the lower left corner, and "My Book" in the lower right corner.

2.3. Double Spacing

NROFF will double space output text automatically if you use the request `.ls 2`, as is done in this section. You can revert to single spaced mode by typing `.ls 1`.

2.4. Page Layout

A number of requests allow you to change the way the printed copy looks, sometimes called the *layout* of the output page. Most of these requests adjust the placing of "white space" (blank lines or spaces). In these explanations, characters in italics should be replaced with values you wish to use; bold characters represent characters which should actually be typed.

The `.bp` request starts a new page.

The request `.sp N` leaves *N* lines of blank space. *N* can be omitted (meaning skip a single line) or can be of the form *Ni* (for *N* inches) or *Nc* (for *N* centimeters). For example, the input:

```
.sp 1.5i
My thoughts on the subject
.sp
```

leaves one and a half inches of space, followed by the line "My thoughts on the subject", followed by a single blank line.

The `.in +N` request changes the amount of white space on the left of the page (the *indent*). The argument *N* can be of the form `+N` (meaning leave *N* spaces more than you are already

```
.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

```
.(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.

The .xp request prints the index.

For example, the input:

```
.(x
  Sealing wax
.)x
.(x
  Cabbages and kings
.)x _
.(x
  Why the sea is boiling hot
.)x 2.5a
.(x
  Whether pigs have wings
.)x ""
.(x
  This is a terribly long index entry, such as might be used
  for a list of illustrations, tables, or figures; I expect it to
  take at least two lines.
.)x
.xp
```

generates:

Sealing wax	9
Cabbages and kings	
Why the sea is boiling hot	2.5a
Whether pigs have wings	
This is a terribly long index entry, such as might be used for a list of illustrations, tables, or figures; I expect it to take at least two lines.	9

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 .lp request. A word specified on the same line as .lp is printed in the margin, and the body is lined up at a prespecified position (normally five spaces). 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
```

produces³:

- One egg yolk

³By 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.

which will do a section heading, but will put no number on the section.

5.3. Parts of the Basic Paper

There are some requests which assist in setting up papers. The `.tp` request initializes for a title page. There are no headers or footers on a title page, and unlike other pages you can space down and leave blank space at the top. For example, a typical title page might appear as:

```
.tp
.sp 2i
.(l C
THE GROWTH OF TOENAILS
IN UPPER PRIMATES
.sp
by
.sp
Frank N. Furter
.)l
.bp
```

The request `.th` sets up the environment of the NROFF processor to do a thesis, using the rules established at Berkeley. It defines the correct headers and footers (a page number in the upper right hand corner only), sets the margins correctly, and double spaces.

The `+.c T` request can be used to start chapters. Each chapter is automatically numbered from one, and a heading is printed at the top of each chapter with the chapter number and the chapter name *T*. For example, to begin a chapter called "Conclusions", use the request:

```
+.c "CONCLUSIONS"
```

which will produce, on a new page, the lines

```
CHAPTER 5
CONCLUSIONS
```

with appropriate spacing for a thesis. Also, the header is moved to the foot of the page on the first page of a chapter. Although the `+.c` request was not designed to work only with the `.th` request, it is tuned for the format acceptable for a PhD thesis at Berkeley.

If the title parameter *T* is omitted from the `+.c` request, the result is a chapter with no heading. This can also be used at the beginning of a paper; for example, `+.c` was used to generate page one of this document.

Although papers traditionally have the abstract, table of contents, and so forth at the front of the paper, it is more convenient to format and print them last when using NROFF. This is so that index entries can be collected and then printed for the table of contents (or whatever). At the end of the paper, issue the `++. P` request, which begins the preliminary part of the paper. After issuing this request, the `+.c` request will begin a preliminary section of the paper. Most notably, this prints the page number restarted from one in lower case Roman numbers. `+.c` may be used repeatedly to begin different parts of the front material for example, the abstract, the table of contents, acknowledgments, list of illustrations, etc. The request `++. B` may also be used to begin the bibliographic section at the end of the paper. For example, the paper might appear as outlined in figure 2. (In this figure, comments begin with the sequence `\`.)

5.4. Equations and Tables

Two special UNIX programs exist to format special types of material. `Eqn` and `neqn` set equations for the phototypesetter and NROFF respectively. `Tbl` arranges to print extremely



The `.EQ` request may take an equation number as an optional argument, which is printed vertically centered on the right hand side of the equation. If the equation becomes too long it should be split between two lines. To do this, type:

```
.EQ (eq 34)
text of equation 34
.EN C
.EQ
continuation of equation 34
.EN
```

The `C` on the `.EN` request specifies that the equation will be continued.

The `tbl` program produces tables. It is fully described (including numerous examples) in the document *Tbl - A Program to Format Tables* by M. E. Lesk. Tables begin with the `.TS` request and end with the `.TE` request. Tables are normally kept on a single page. If you have a table which is too big to fit on a single page, so that you know it will extend to several pages, begin the table with the request `.TS H` and put the request `.TH` after the part of the table which you want duplicated at the top of every page that the table is printed on. For example, a table definition for a long table might look like:

```
.TS H
c s s
n n n.
THE TABLE TITLE
.TH
text of the table
.TE
```

5.5. Two Column Output

You can get two column output automatically by using the request `.2c`. This causes everything after it to be output in two-column form. The request `.bc` will start a new column; it differs from `.bp` in that `.bp` may leave a totally blank column when it starts a new page. To revert to single column output, use `.1c`.

5.6. Defining Macros

A *macro* is a collection of requests and text which may be used by stating a simple request. Macros begin with the line `.de xx` (where `xx` is the name of the macro to be defined) and end with the line consisting of two dots. After defining the macro, stating the line `.xx` is the same as stating all the other lines. For example, to define a macro that spaces 3 lines and then centers the next input line, enter:

```
.de SS
.sp 3
.ce
..
```

and use it by typing:

```
.SS
Title Line
(beginning of text)
```

Macro names may be one or two characters. In order to avoid conflicts with names in `-me`, always use upper case letters as names. The only names to avoid are `TS`, `TH`, `TE`, `EQ`, and `EN`.

to appear, you should quote the entire string (even if a single word), and use *two* quote marks where you want one to appear. For example, if you want to produce the text:

"Master Control"

in italics, you must type:

.i ""Master Control\|""

The \| produces a very narrow space so that the "1" does not overlap the quote sign in TROFF, like this:

"Master Control"

There are also several "pseudo-fonts" available. The input:

```
.(b
.u underlined
.bi "bold italics"
.bx "words in a box"
.)b
```

generates

```
underlined
bold italics
words in a box
```

In NROFF these all just underline the text. Notice that pseudo font requests set only the single parameter in the pseudo font; ordinary font requests will begin setting all text in the special font if you do not provide a parameter. No more than one word should appear with these three font requests in the middle of lines. This is because of the way TROFF justifies text. For example, if you were to issue the requests:

```
.bi "some bold italics"
and
.bx "words in a box"
```

in the middle of a line TROFF would produce ~~some bold italics~~ and words in a box, which I think you will agree does not look good.

The second parameter of all font requests is set in the original font. For example, the font request:

```
.b bold face
```

generates "bold" in bold font, but sets "face" in the font of the surrounding text, resulting in:

```
boldface.
```

To set the two words **bold** and **face** both in **bold face**, type:

```
.b "bold face"
```

You can mix fonts in a word by using the special sequence \c at the end of a line to indicate "continue text processing"; this allows input lines to be joined together without a space between them. For example, the input:

```
.u under \c
.i italics
```

generates under*italics*, but if we had typed:

```
.u under
.i italics
```

the result would have been under *italics* as two words.



I would like to thank Bob Epstein, Bill Joy, and Larry Rowe for having the courage to use the -me macros to produce non-trivial papers during the development stages; Ricki Blau, Pamela Humphrey, and Jim Joyce for their help with the documentation phase; peter kessler for numerous complaints years after I was "done" with this project, most accompanied by fixes (hence forcing me to fix several small bugs); and the plethora of people who have contributed ideas and have given support for the project.

This document was TROFF'ed on April 20, 1986 and applies to version 2.27 of the -me macros.

-ME REFERENCE MANUAL

Release 2.27

Eric P. Allman*

Project INGRES
Electronics Research Laboratory
University of California, Berkeley
Berkeley, California 94720

This document describes in extremely terse form the features of the `-me` macro package for version seven NROFF/TROFF. Some familiarity is assumed with those programs. Specifically, the reader should understand breaks, fonts, point sizes, the use and definition of number registers and strings, how to define macros, and scaling factors for ens, points, v's (vertical line spaces), etc.

For a more casual introduction to text processing using NROFF, refer to the document *Writing Papers with NROFF using -me*.

There are a number of macro parameters that may be adjusted. Fonts may be set to a font number only. Font 8 means bold font in TROFF; in NROFF font 8 is underlined unless the `-rb3` flag is specified to use "true bold" font (most versions of NROFF do not interpret bold font nicely). Font 0 is no font change; the font of the surrounding text is used instead. Notice that fonts 0 and 8 are "pseudo-fonts"; that is, they are simulated by the macros. This means that although it is legal to set a font register to zero or eight, it is not legal to use the escape character form, such as:

```
\f8
```

All distances are in basic units, so it is nearly always necessary to use a scaling factor. For example, the request to set the paragraph indent to eight one-en spaces is:

```
.nr pi 8n
```

and not

```
.nr pi 8
```

which would set the paragraph indent to eight basic units, or about 0.02 inch. Default parameter values are given in brackets in the remainder of this document.

Registers and strings of the form `$x` may be used in expressions but should not be changed. Macros of the form `$x` perform some function (as described) and may be redefined to change this function. This may be a sensitive operation; look at the body of the original macro before changing it.

All names in `-me` follow a rigid naming convention. The user may define number registers, strings, and macros, provided that s/he uses single character upper case names or double character

*Author's current address: Britton Lee, Inc., 1919 Addison Suite 105, Berkeley, California 94704.

†NROFF and TROFF may be trademarks of AT&T Bell Laboratories.



depth, and the section title is extended. (See `.ba`.) Also, an additional indent of `\n(so [0]` is added to the section title (but not to the body of the section). The font is then set to the paragraph font, so that more information may occur on the line with the section number and title. `.sh` insures that there is enough room to print the section head plus the beginning of a paragraph (about 3 lines total). If *a* through *f* are specified, the section number is set to that number rather than incremented automatically. If any of *a* through *f* are a hyphen that number is not reset. If *T* is a single underscore (“_”) then the section depth and numbering is reset, but the base indent is not reset and nothing is printed out. This is useful to automatically coordinate section numbers with chapter numbers.

- `.sx +N` Go to section depth *N* [-1], but do not print the number and title, and do not increment the section number at level *N*. This has the effect of starting a new paragraph at level *N*.
- `.uh T` Unnumbered section heading. The title *T* is printed with the same rules for spacing, font, etc., as for `.sh`.
- `.$p T B N` Print section heading. May be redefined to get fancier headings. *T* is the title passed on the `.sh` or `.uh` line; *B* is the section number for this section, and *N* is the depth of this section. These parameters are not always present; in particular, `.sh` passes all three, `.uh` passes only the first, and `.sx` passes three, but the first two are null strings. Care should be taken if this macro is redefined; it is quite complex and subtle.
- `.$0 T B N` This macro is called automatically after every call to `.$p`. It is normally undefined, but may be used to automatically put every section title into the table of contents or for some similar function. *T* is the section title for the section title which was just printed, *B* is the section number, and *N* is the section depth.
- `.$1 - .$.6` Traps called just before printing that depth section. May be defined to (for example) give variable spacing before sections. These macros are called from `.$p`, so if you redefine that macro you may lose this feature.

3. Headers and Footers

Headers and footers are put at the top and bottom of every page automatically. They are set in font `\n(tf [3]` and size `\n(tp [10p]`. Each of the definitions apply as of the *next* page. Three-part titles must be quoted if there are two blanks adjacent anywhere in the title or more than eight blanks total.

The spacing of headers and footers are controlled by three number registers. `\n(hm [4v]` is the distance from the top of the page to the top of the header, `\n(fm [3v]` is the distance from the bottom of the page to the bottom of the footer, `\n(tm [7v]` is the distance from the top of the page to the top of the text, and `\n(bm [6v]` is the distance from the bottom of the page to the bottom of the text (nominal). The macros `.m1`, `.m2`, `.m3`, and `.m4` are also supplied for compatibility with ROFF documents.

- `.he l'm r'` Define three-part header, to be printed on the top of every page.
- `.fo l'm r'` Define footer, to be printed at the bottom of every page.
- `.eh l'm r'` Define header, to be printed at the top of every even-numbered page.
- `.oh l'm r'` Define header, to be printed at the top of every odd-numbered page.
- `.ef l'm r'` Define footer, to be printed at the bottom of every even-numbered page.
- `.of l'm r'` Define footer, to be printed at the bottom of every odd-numbered page.

text changes. The floating keep is preceded and followed by `\n(zs [1v]` space. Also, it defaults to mode `M`.

- `.)z` End floating keep.
- `.(c` Begin centered block. The next keep is centered as a block, rather than on a line-by-line basis as with `.(b C`. This call may be nested inside keeps.
- `.)c` End centered block.

5. Annotations

- `.(d` Begin delayed text. Everything in the next keep is saved for output later with `.pd`, in a manner similar to footnotes.
- `.)d n` End delayed text. The delayed text number register `\n($d` and the associated string `**` are incremented if `**` has been referenced.
- `.pd` Print delayed text. Everything diverted via `.(d` is printed and truncated. This might be used at the end of each chapter.
- `.(f` Begin footnote. The text of the footnote is floated to the bottom of the page and set in font `\n(ff [1]` and size `\n(fp [8p]`. Each entry is preceded by `\n(fs [0.2v]` space, is indented `\n(fi [3n]` on the first line, and is indented `\n(fu [0]` from the right margin. Footnotes line up underneath two column output. If the text of the footnote will not all fit on one page it will be carried over to the next page.
- `.)f n` End footnote. The number register `\n($f` and the associated string `**` are incremented if they have been referenced.
- `.$s` The macro to output the footnote separator. This macro may be redefined to give other size lines or other types of separators. Currently it draws a 1.5i line.
- `.(x x` Begin index entry. Index entries are saved in the index `x [x]` until called up with `.xp`. Each entry is preceded by a `\n(xs [0.2v]` space. Each entry is “undented” by `\n(xu [0.5i]`; this register tells how far the page number extends into the right margin.
- `.)x P A` End index entry. The index entry is finished with a row of dots with `A [null]` right justified on the last line (such as for an author’s name), followed by `P [\n%]`. If `A` is specified, `P` must be specified; `\n%` can be used to print the current page number. If `P` is an underscore, no page number and no row of dots are printed.
- `.xp x` Print index `x [x]`. The index is formatted in the font, size, and so forth in effect at the time it is printed, rather than at the time it is collected.

6. Columned Output

- `.2c +S N` Enter two-column mode. The column separation is set to `+S [4n, 0.5i]` in ACM mode] (saved in `\n($s)`. The column width, calculated to fill the single column line length with both columns, is stored in `\n($l`. The current column is in `\n($c`. You can test register `\n($m [1]` to see if you are in single column or double column mode. Actually, the request enters `N [2]` column output.
- `.1c` Revert to single-column mode.
- `.bc` Begin column. This is like `.bp` except that it begins a new column on a new page only if necessary, rather than forcing a whole new page if there is another column left on the current page.

no space will ever be output.

9. Preprocessor Support

- .EQ *m T*** Begin equation. The equation is centered if *m* is C or omitted, indented \n[bi [4m] if *m* is I, and left justified if *m* is L. *T* is a title printed on the right margin next to the equation. See *Typesetting Mathematics - User's Guide* by Brian W. Kernighan and Lorinda L. Cherry.
- .EN *c*** End equation. If *c* is C the equation must be continued by immediately following with another .EQ, the text of which can be centered along with this one. Otherwise, the equation is printed, always on one page, with \n(es [0.5v in TROFF, 1v in NROFF] space above and below it.
- .TS *h*** Table start. Tables are single spaced and kept on one page if possible. If you have a large table which will not fit on one page, use *h* = H and follow the header part (to be printed on every page of the table) with a .TH. See *Tbl - A Program to Format Tables* by M. E. Lesk.
- .TH** With .TS H, ends the header portion of the table.
- .TE** Table end. Note that this table does not float, in fact, it is not even guaranteed to stay on one page if you use requests such as .sp intermixed with the text of the table. If you want it to float (or if you use requests inside the table), surround the entire table (including the .TS and .TE requests) with the requests .(z and).z.
- .PS *h w*** Begin *pic* picture. *H* is the height and *w* is the width, both in basic units. *Ditroff* only.
- .PE** End picture.
- .IS** Begin *ideal* picture.
- .IE** End *ideal* picture.
- .IF** End *ideal* picture (alternate form).
- .GS** Begin *gremlin* picture.
- .GE** End *gremlin* picture.
- .GF** End *gremlin* picture (alternate form).

10. Miscellaneous

- .re** Reset tabs. Set to every 0.5i in TROFF and every 0.8i in NROFF.
- .ba +*N*** Set the base indent to +*N* [0] (saved in \n(\$i). All paragraphs, sections, and displays come out indented by this amount. Titles and footnotes are unaffected. The .sh request performs a .ba request if \n(si [0] is not zero, and sets the base indent to \n(si*\n(\$0).
- .xl +*N*** Set the line length to *N* [6.0i]. This differs from .ll because it only affects the current environment.
- .ll +*N*** Set line length in all environments to *N* [6.0i]. This should not be used after output has begun, and particularly not in two-column output. The current line length is stored in \n(\$l).
- .hl** Draws a horizontal line the length of the page. This is useful inside floating keeps to differentiate between the text and the figure.
- .lh** Print a letterhead at the current position on the page. The format of the letterhead must be defined in the file /usr/lib/me/letterhead.me by your local



part which will be printed in the conference proceedings), together with the current page number and the total number of pages *N*. Additionally, this macro loads the file `/usr/lib/me/acm.me`, which may later be augmented with other macros useful for printing papers for ACM conferences. It should be noted that this macro will not work correctly in version 7 TROFF, since it sets the page length wider than the physical width of the C/A/T phototypesetter roll.

12. Predefined Strings

- `**` Footnote number, actually `*[n($f*)]`. This macro is incremented after each call to `.f`.
- `*#` Delayed text number. Actually `[n($d)]`.
- `*[` Superscript. This string gives upward movement and a change to a smaller point size if possible, otherwise it gives the left bracket character ('['). Extra space is left above the line to allow room for the superscript.
- `*]` Unsuperscript. Inverse to `*[`. For example, to produce a superscript you might type `x\[2*]`, which will produce x^2 .
- `*<` Subscript. Defaults to '<' if half-carriage motion not possible. Extra space is left below the line to allow for the subscript.
- `*>` Inverse to `*<`.
- `*(dw` The day of the week, as a word.
- `*(mo` The month, as a word.
- `*(td` Today's date, directly printable. The date is of the form April 20, 1986. Other forms of the date can be used by using `\n(dy` (the day of the month; for example, 20), `*(mo` (as noted above) or `\n(mo` (the same, but as an ordinal number; for example, April is 4), and `\n(yr` (the last two digits of the current year).
- `*(lq` Left quote marks. Double quote in NROFF.
- `*(rq` Right quote.
- `*-` ¾ em dash in TROFF; two hyphens in NROFF.

13. Special Characters and Marks

There are a number of special characters and diacritical marks (such as accents) available through `-me`. To reference these characters, you must call the macro `.sc` to define the characters before using them.

`.sc` Define special characters and diacritical marks, as described in the remainder of this section. This macro must be stated before initialization. The special characters available are listed below.

Name	Usage	Example	
Acute accent	<code>*</code>	<code>a*</code>	á
Grave accent	<code>*</code>	<code>e*</code>	è
Umlat	<code>*.</code>	<code>u*.</code>	ü
Tilde	<code>*~</code>	<code>n*~</code>	ñ
Caret	<code>*^</code>	<code>e*^</code>	ê
Cedilla	<code>*,</code>	<code>c*,</code>	ç
Czech	<code>*v</code>	<code>e*v</code>	ě
Circle	<code>*o</code>	<code>A*o</code>	Å
There exists	<code>*(qe</code>		∃



Summary

This alphabetical list summarizes all macros, strings, and number registers available in the `-me` macros. Selected *troff* commands, registers, and functions are included as well; those listed can generally be used with impunity.

The columns are the name of the command, macro, register, or string; the type of the object, and the description. Types are **M** for macro or builtin command (invoked with `.` or `'` in the first input column), **S** for a string (invoked with `*` or `*(`), **R** for a number register (invoked with `\n` or `\n()`), and **F** for a *troff* builtin function (invoked by preceding it with a single backslash).

Lines marked with \S are *troff* internal codes. Lines marked with \dagger or \ddagger may be defined by the user to get special functions; \ddagger indicates that these are defined by default and changing them may have unexpected side effects. Lines marked with $^\circ$ are specific to *ditroff* (device-independent *troff*).

NAME	TYPE	DESCRIPTION
<code>\(space)</code>	F \S	unpaddable space
<code>\"</code>	F \S	comment (to end of line)
<code>*#</code>	S	optional delayed text tag string
<code>\\$N</code>	F \S	interpolate argument <i>N</i>
<code>\n(\$0)</code>	R	section depth
<code>.\$0</code>	M \dagger	invoked after section title printed
<code>\n(\$1)</code>	R	first section number
<code>.\$1</code>	M \dagger	invoked before printing depth 1 section
<code>\n(\$2)</code>	R	second section number
<code>.\$2</code>	M \dagger	invoked before printing depth 2 section
<code>\n(\$3)</code>	R	third section number
<code>.\$3</code>	M \dagger	invoked before printing depth 3 section
<code>\n(\$4)</code>	R	fourth section number
<code>.\$4</code>	M \dagger	invoked before printing depth 4 section
<code>\n(\$5)</code>	R	fifth section number
<code>.\$5</code>	M \dagger	invoked before printing depth 5 section
<code>\n(\$6)</code>	R	sixth section number
<code>.\$6</code>	M \dagger	invoked before printing depth 6 section
<code>.\$C</code>	M \dagger	called at beginning of chapter
<code>.\$H</code>	M \dagger	text header
<code>\n(\$R)</code>	R \ddagger	relative vertical spacing in displays
<code>\n(\$c)</code>	R	current column number
<code>.\$c</code>	M \ddagger	print chapter title
<code>\n(\$d)</code>	R	delayed text number
<code>\n(\$f)</code>	R	footnote number
<code>.\$f</code>	M \ddagger	print footer
<code>.\$h</code>	M \ddagger	print header
<code>\n(\$i)</code>	R	paragraph base indent
<code>\n(\$l)</code>	R	column width
<code>\n(\$m)</code>	R	number of columns in effect
<code>*(\$n)</code>	S	section name
<code>\n(\$p)</code>	R	numbered paragraph number
<code>.\$p</code>	M \ddagger	print section heading (internal macro)
<code>\n(\$r)</code>	R \ddagger	relative vertical spacing in text
<code>\n(\$s)</code>	R	column indent
<code>.\$s</code>	M \ddagger	footnote separator (from text)
<code>\n%</code>	R \S	current page number
<code>\&</code>	F \S	zero width character, useful for hiding controls
<code>\(xx)</code>	F \S	interpolate special character <i>xx</i>
<code>.(b</code>	M	begin block

NAME	TYPE	DESCRIPTION
\{	F§	grave accent
*]	S	end superscript
\~	F§	1/12 em narrow space
*^	S	caret
.ac	M	ACM mode
.ad	M§	set text adjustment
.af	M§	assign format to register
.am	M§	append to macro
.ar	M	set page numbers in Arabic
.as	M§	append to string
.b	M	bold font
.ba	M	set base indent
.bc	M	begin new column
.bi	M	bold italic
\n(bi	R	display (block) indent
.bl	M	blank lines (even at top of page)
\n(bm	R	bottom title margin
.bp	M§	begin page
.br	M§	break (start new line)
\n(bs	R	display (block) pre/post spacing
\n(bt	R	block keep threshold
.bx	M	boxed
\c	F§	continue input
.ce	M§	center lines
\n(ch	R	current chapter number
.de	M§	define macro
\n(df	R	display font
.ds	M§	define string
\n(dw	R§	current day of week
*(dw	S	current day of week
\n(dy	R§	day of month
\e	F§	printable version of \
.ef	M	set footer (even numbered pages only)
.eh	M	set header (even numbered pages only)
.el	M§	else part of conditional
.ep	M	end page
\n(es	R	equation pre/post space
\f	F§	inline font change to font <i>f</i>
\f <i>ff</i>	F§	inline font change to font <i>ff</i>
.fc	M§	set field characters
\n(ff	R	footnote font
.fi	M§	fill output lines
\n(fi	R	footnote indent (first line only)
\n(fm	R	footer margin
.fo	M	set footer
\n(fp	R	footnote pointsize
\n(fs	R	footnote prespace
\n(fu	R	footnote undent (from right margin)
\h <i>d</i>	F§	local horizontal motion for distance <i>d</i>
.hc	M§	set hyphenation character
.he	M	set header
.hl	M	draw horizontal line



NAME	TYPE	DESCRIPTION
*(qa)	S	for all
*(qe)	S	there exists
\n(qi)	R	quote indent (also shortens line)
\n(qp)	R	quote pointsize
\n(qs)	R	quote pre/post space
.r	M	roman font
.rb	M	real bold font
.re	M	reset tabs
.rm	M§	remove macro or string
.rn	M§	rename macro or string
.ro	M	set page numbers in roman
*(rq)	S	right quote marks
.rr	M§	remove register
.rs	M§	restore spacing
.rt	M§	return to vertical position
\sS	F§	inline size change to size S
.sc	M	load special characters
\n(sf)	R	section title font
.sh	M	begin numbered section
\n(si)	R	relative base indent per section depth
.sk	M	skip next page
.sm	M	set argument in a smaller pointsize
.so	M§	source input file
\n(so)	R	additional section title offset
.sp	M§	vertical space
\n(sp)	R	section title pointsize
\n(ss)	R	section prespace
.sx	M	change section depth
.sz	M	set pointsize and vertical spacing
.ta	M§	set tab stops
.tc	M§	set tab repetition character
*(td)	S	today's date
\n(tf)	R	title font
.th	M	set thesis mode
.ti	M§	temporary indent (next line only)
.tl	M§	three part title
\n(tm)	R	top title margin
.tp	M	begin title page
\n(tp)	R	title pointsize
.tr	M§	translate
.u	M	underlined
.uh	M	unnumbered section
.ul	M§	underline next line
\v d	F§	local vertical motion for distance d
*v	S	inverted 'v' for czeck "ě"
\w S	F§	return width of string S
.xl	M	set line length (local)
.xp	M	print index
\n(xs)	R	index entry prespace
\n(xu)	R	index undent (from right margin)
\n(yr)	R§	year (last two digits only)
\n(zs)	R	floating keep pre/post space

NROFF/TROFF User's Manual

Joseph F. Ossanna
(updated for 4.3BSD by Mark Seiden)

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

NROFF and TROFF are text processors under the UNIX Time-Sharing System 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.

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.
-mname	Prepends the macro file <code>/usr/lib/tmac.name</code> to the input <i>files</i> .
-nN	Number first generated page <i>N</i> .
-olist	Print only pages whose page numbers appear in <i>list</i> , which consists of comma-separated numbers and number ranges. A number range has the form <i>N-M</i> and means pages <i>N</i> through <i>M</i> ; a initial <i>-N</i> means from the beginning to page <i>N</i> ; and a final <i>N-</i> means from <i>N</i> to the end.
-q	Invoke the simultaneous input-output mode of the rd request.
-raN	Number register <i>a</i> (one-character) is set to <i>N</i> .
-sN	Stop every <i>N</i> pages. NROFF will halt prior to every <i>N</i> pages (default <i>N</i> =1) to allow paper loading or changing, and will resume upon receipt of a newline. TROFF will stop the phototypesetter every <i>N</i> 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).

SUMMARY OF REQUESTS AND OUTLINE OF THIS MANUAL

<i>Request Form</i>	<i>Initial Value*</i>	<i>If No Argument</i>	<i>Notes#</i>	<i>Explanation</i>
1. General Explanation				
2. Font and Character Size Control				
.ps ±N	10 point	previous	E	Point size; also \s±N.†
.fz F ±N	off	-	E	font F to point size ±N.
.fz S F ±N	off	-	E	Special Font characters to point size ±N.
.ss N	12/36 em	ignored	E	Space-character size set to N/36 em.†
.cs FNM	off	-	P	Constant character space (width) mode (font F).†
.bd F N	off	-	P	Embolden font F by N-1 units.†
.bd S F N	off	-	P	Embolden Special Font when current font is F.†
.ft F	Roman	previous	E	Change to font F = x, xx, or 1-4. Also \lx, \f(xx, \fN.
.fp N F	R,1,B,S	ignored	-	Font named F mounted on physical position 1≤N≤4.

3. Page Control				
.pl ±N	11 in	11 in	v	Page length.
.bp ±N	N=1	-	B‡,v	Eject current page; next page number N.
.pn ±N	N=1	ignored	-	Next page number N.
.po ±N	0; 26/27 in	previous	v	Page offset.
.ne N	-	N=1V	D,v	Need N vertical space (V = vertical spacing).
.mk R	none	internal	D	Mark current vertical place in register R.
.rt ±N	none	internal	D,v	Return (<i>upward only</i>) to marked vertical place.

4. Text Filling, Adjusting, and Centering				
.br	-	-	B	Break.
.fi	fill	-	B,E	Fill output lines.
.nf	fill	-	B,E	No filling or adjusting of output lines.
.ad c	adj,both	adjust	E	Adjust output lines with mode c.
.na	adjust	-	E	No output line adjusting.
.ce N	off	N=1	B,E	Center following N input text lines.

5. Vertical Spacing				
.vs N	1/6in;12pts	previous	E,p	Vertical base line spacing (V).
.ls N	N=1	previous	E	Output N-1 V's after each text output line.
.sp N	-	N=1V	B,v	Space vertical distance N in either direction.
.sv N	-	N=1V	v	Save vertical distance N.
.os	-	-	-	Output saved vertical distance.
.ns	space	-	D	Turn no-space mode on.
.rs	-	-	D	Restore spacing; turn no-space mode off.

6. Line Length and Indenting				
.ll ±N	6.5 in	previous	E,m	Line length.
.in ±N	N=0	previous	B,E,m	Indent.
.ti ±N	-	ignored	B,E,m	Temporary indent.

7. Macros, Strings, Diversion, and Position Traps				
.de xx yy	-	.yy=..	-	Define or redefine macro xx; end at call of yy.
.am xx yy	-	.yy=..	-	Append to a macro.

*Values separated by ";" are for NROFF and TROFF respectively.
 #Notes are explained at the end of this Summary and Index
 †No effect in NROFF.
 ‡The use of " " as control character (instead of ".") suppresses the break function.



<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
---------------------	----------------------	-----------------------	--------------	--------------------

16. Conditional Acceptance of Input

<code>.if c anything</code>	-	-		If condition <i>c</i> true, accept <i>anything</i> as input, for multi-line use <code>\(anything\)</code> .
<code>.if !c anything</code>	-	-		If condition <i>c</i> false, accept <i>anything</i> .
<code>.if N anything</code>	-	u		If expression <i>N</i> > 0, accept <i>anything</i> .
<code>.if !N anything</code>	-	u		If expression <i>N</i> ≤ 0, accept <i>anything</i> .
<code>.if 'string1 'string2' anything</code>	-	-		If <i>string1</i> identical to <i>string2</i> , accept <i>anything</i> .
<code>.if ! 'string1 'string2' anything</code>	-	-		If <i>string1</i> not identical to <i>string2</i> , accept <i>anything</i> .
<code>.ie c anything</code>	-	u		If portion of if-else; all above forms (like if).
<code>.el anything</code>	-	-		Else portion of if-else.

17. Environment Switching.

<code>.ev N</code>	<i>N</i> =0	previous	-	Environment switched (<i>push down</i>).
--------------------	-------------	----------	---	--

18. Insertions from the Standard Input

<code>.rd prompt</code>	-	<i>prompt</i> =BEL		Read insertion.
<code>.ex</code>	-	-	-	Exit from NROFF/TROFF.

19. Input/Output File Switching

<code>.so filename</code>	-	-		Switch source file (<i>push down</i>).
<code>.nx filename</code>		end-of-file	-	Next file.
<code>.pi program</code>	-	-		Pipe output to <i>program</i> (NROFF only).

20. Miscellaneous

<code>.mc c N</code>	-	off	E,m	Set margin character <i>c</i> and separation <i>N</i> .
<code>.tm string</code>	-	newline	-	Print <i>string</i> on terminal (UNIX standard error output).
<code>.ig yy</code>	-	.yy=..	-	Ignore till call of <i>yy</i> .
<code>.pm t</code>	-	all	-	Print macro names and sizes; if <i>t</i> present, print only total of sizes.
<code>.ab string</code>	-	-	-	Print a message and abort.
<code>.fl</code>	-	-	B	Flush output buffer.

21. Output and Error Messages

Notes-

- B Request normally causes a break.
- D Mode or relevant parameters associated with current diversion level.
- E Relevant parameters are a part of the current environment.
- O Must stay in effect until logical output.
- P Mode must be still or again in effect at the time of physical output.
- v,p,m,u Default scale indicator; if not specified, scale indicators are *ignored*.

Alphabetical Request and Section Number Cross Reference

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	nm15	pl 3	rr 8	tc 9	wh 7
as 7	cs 2	el 16	fp 2	ig 20	lt 14	nn 15	pm20	rs 5	ti 6	
bd 2	cu 10	em 7	ft 2	in 6	mc20	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	lc 9	na 4	nx 19	ps 2	sp 5	tr 10	

Predefined General Number Registers

<i>Section Reference</i>	<i>Register Name</i>	<i>Description</i>
3	%	Current page number.
19	c.	Number of <i>lines</i> read from current input file.
11.2	ct	Character type (set by <i>width</i> 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 <i>input</i> 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 <i>width</i> function).
11.2	st	Height of string above base line (generated by <i>width</i> function).
-	yr	Last two digits of current year.

Predefined Read-Only Number Registers

<i>Section Reference</i>	<i>Register Name</i>	<i>Description</i>
7.3	.\$	Number of arguments available at the current macro level.
-	.A	Set to 1 in TROFF, if <i>-a</i> option used; always 1 in NROFF.
11.1	.H	Available horizontal resolution in basic units.
5.3	.L	Set to current <i>line-spacing</i> (ls) parameter
-	.P	Set to 1 if the current page is being printed; otherwise 0.
-	.T	Set to 1 in NROFF, if <i>-T</i> 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 <i>\x'N'</i> .
19	.c	Number of <i>lines</i> read from current input file.
7.4	.d	Current vertical place in current diversion; equal to <i>nl</i> , 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.

the *input* line to the horizontal place *N*. For example,

```
.sp | 3.2c
```

will space *in the required direction* to 3.2 centimeters from the top of the page.

1.4. Numerical expressions. Wherever numerical input is expected, an expression involving parentheses, the arithmetic operators +, -, /, *, % (mod), and the logical operators <, >, <=, >=, = (or ==), & (and), : (or) may be used. Except where controlled by parentheses, evaluation of expressions is left-to-right; there is no operator precedence. In the case of certain requests, an initial + or - is stripped and interpreted as an increment or decrement indicator respectively. In the presence of default scaling, the desired scale indicator must be attached to *every* number in an expression for which the desired and default scaling differ. For example, if the number register *x* contains 2 and the current point size is 10, then

```
.ll (4.25i+\nxP+3)/2u
```

will set the line length to 1/2 the sum of 4.25 inches + 2 picas + 30 points.

1.5. Notation. Numerical parameters are indicated in this manual in two ways. ±*N* means that the argument may take the forms *N*, +*N*, or -*N* and that the corresponding effect is to set the affected parameter to *N*, to increment it by *N*, or to decrement it by *N* respectively. Plain *N* means that an initial algebraic sign is *not* an increment indicator, but merely the sign of *N*. Generally, unreasonable numerical input is either ignored or truncated to a reasonable value. For example, most requests expect to set parameters to non-negative values; exceptions are **sp**, **wh**, **ch**, **nr**, and **if**. The requests **ps**, **ft**, **po**, **vs**, **ls**, **ll**, **in**, and **l** 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*xx*, \f*N* where *x* and *xx* are the name of a mounted font and *N* is a numerical font position. It is *not*



were printed with **.bd I 3**. The mode must be still or again in effect when the characters are physically printed. Ignored in NROFF.

.bd <i>S F N</i>	off	-	P	The characters in the Special Font will be emboldened whenever the current font is <i>F</i> . This manual was printed with .bd SB 3 . The mode must be still or again in effect when the characters are physically printed.
.ft <i>F</i>	Roman	previous	E	Font changed to <i>F</i> . Alternatively, imbed \MF . The font name <i>P</i> is reserved to mean the previous font.
.fp <i>N F</i>	R,I,B,S	ignored	-	Font position. This is a statement that a font named <i>F</i> is mounted on position <i>N</i> (1-4). It is a fatal error if <i>F</i> is not known. The 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.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.pl $\pm N$	11 in	11 in	v	Page length set to $\pm N$. The internal limitation is about 75 inches in TROFF and about 136 inches in NROFF. The current page length is available in the .p register.
.bp $\pm N$	<i>N</i> =1	-	B*,v	Begin page. The current page is ejected and a new page is begun. If $\pm N$ is given, the new page number will be $\pm N$. Also see request ns.
.pn $\pm N$	<i>N</i> =1	ignored	-	Page number. The next page (when it occurs) will have the page number $\pm N$. A pn must occur before the initial pseudo-page transition to affect the page number of the first page. The current page number is in the % register.
.po $\pm N$	0; 26/27 in†	previous	v	Page offset. The current <i>left margin</i> is set to $\pm N$. The TROFF initial value provides about 1 inch of paper margin including the physical typesetter margin of 1/27 inch. In TROFF the maximum (line-length)+(page-offset) is about 7.54 inches. See §6. The current page offset is available in the .o register.
.ne <i>N</i>	-	<i>N</i> =1 <i>V</i>	D,v	Need <i>N</i> vertical space. If the distance, <i>D</i> , to the next trap position (see §7.5) is less than <i>N</i> , a forward vertical

*The use of " " as control character (instead of ".") suppresses the break function.

†Values separated by ";" are for NROFF and TROFF respectively.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.br	-	-	B	Break. The filling of the line currently being collected is stopped and the line is output without adjustment. Text lines beginning with space characters and empty text lines (blank lines) also cause a break.
.fi	fill on	-	B,E	Fill subsequent output lines. The register .u is 1 in fill mode and 0 in nofill mode.
.nf	fill on	-	B,E	Nofill. Subsequent output lines are <i>neither filled nor</i> adjusted. Input text lines are copied directly to output lines <i>without regard</i> for the current line length.
.ad <i>c</i>	adj,both	adjust	E	Line adjustment is begun. If fill mode is not on, adjustment will be deferred until fill mode is back on. If the type indicator <i>c</i> 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 <i>j</i> number register, which is set to contain the current adjustment mode and type.

Indicator	Adjust Type
l	adjust left margin only
r	adjust right margin only
c	center
b or n	adjust both margins
absent	unchanged

.na	adjust	-	E	Noadjust. Adjustment is turned off; the right margin will be ragged. The adjustment type for <i>ad</i> is not changed. Output line filling still occurs if fill mode is on.
.ce <i>N</i>	off	<i>N</i> =1	B,E	Center the next <i>N</i> input text lines within the current (line-length minus indent). If <i>N</i> =0, any residual count is cleared. A break occurs after each of the <i>N</i> input lines. If the input line is too long, it will be left adjusted.

5. Vertical Spacing

5.1. Base-line spacing. The vertical spacing (*V*) between the base-lines of successive output lines can be set using the vs request with a resolution of 1/144 inch = 1/2 point in TROFF, and to the output device resolution in NROFF. *V* must be large enough to accommodate the character sizes on the affected output lines. For the common type sizes (9-12 points), usual typesetting practice is to set *V* to 2 points greater than the point size; TROFF default is 10-point type on a 12-point spacing (as in this document). The current *V* is available in the .v register. Multiple-*V* line separation (e.g. double spacing) may be requested with ls.

5.2. Extra line-space. If a word contains a vertically tall construct requiring the output line containing it to have extra vertical space before and/or after it, the extra-line-space function \x'*N*' can be imbedded in or attached to that word. In this and other functions having a pair of delimiters around their parameter (here `'), the delimiter choice is arbitrary, except that it can't look like the continuation of a number expression for *N*. If *N* is negative, the output line containing the word will be preceded by *N* extra vertical space; if *N* is positive, the output line containing the word will be followed by *N* extra vertical space. If successive requests for extra space apply to the same line, the maximum values are used. The most recently utilized post-line extra line-space is available in the .a register.



`.ti ±N` - ignored B,E,m Temporary indent. The *next* output text line will be indented a distance $\pm N$ with respect to the current indent. The resulting total indent may not be negative. The current indent is not changed.

7. Macros, Strings, Diversion, and Position Traps

7.1. Macros and strings. A *macro* is a named set of arbitrary *lines* that may be invoked by name or with a *trap*. A *string* is a named string of *characters*, *not* including a newline character, that may be interpolated by name at any point. Request, macro, and string names share the *same* name list. Macro and string names may be one or two characters long and may usurp previously defined request, macro, or string names. Any of these entities may be renamed with `rn` or removed with `rm`. Macros are created by `de` and `di`, and appended to by `am` and `da`; `di` and `da` cause normal output to be stored in a macro. Strings are created by `ds` and appended to by `as`. A macro is invoked in the same way as a request; a control line beginning `.xx` will interpolate the contents of macro `xx`. The remainder of the line may contain up to nine *arguments*. The strings `x` and `xx` are interpolated at any desired point with `*x` and `*(xx` respectively. String references and macro invocations may be nested.

7.2. Copy mode input interpretation. During the definition and extension of strings and macros (not by diversion) the input is read in *copy mode*. The input is copied without interpretation *except* that:

- The contents of number registers indicated by `\n` are interpolated.
- Strings indicated by `*` are interpolated.
- Arguments indicated by `\$` are interpolated.
- Concealed newlines indicated by `\(newline)` are eliminated.
- Comments indicated by `*` are eliminated.
- `\t` and `\a` are interpreted as ASCII horizontal tab and SOH respectively (§9).
- `\\` is interpreted as `\`.
- `\.` is interpreted as `."`.

These interpretations can be suppressed by prepending a `\`. For example, since `\\` maps into a `\`, `\\n` will copy as `\n` which will be interpreted as a number register indicator when the macro or string is reread.

7.3. Arguments. When a macro is invoked by name, the remainder of the line is taken to contain up to nine arguments. The argument separator is the space character, and arguments may be surrounded by double-quotes to permit imbedded space characters. Pairs of double-quotes may be imbedded in double-quoted arguments to represent a single double-quote. If the desired arguments won't fit on a line, a concealed newline may be used to continue on the next line.

When a macro is invoked the *input level* is *pushed down* and any arguments available at the previous level become unavailable until the macro is completely read and the previous level is restored. A macro's own arguments can be interpolated at *any* point within the macro with `\$N`, which interpolates the *N*th argument ($1 \leq N \leq 9$). 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.

				blanks.
<code>.as <i>xx string</i></code>	-	ignored	-	Append <i>string</i> to string <i>xx</i> (append version of <code>ds</code>).
<code>.rm <i>xx</i></code>	-	ignored	-	Remove request, macro, or string. The name <i>xx</i> is removed from the name list and any related storage space is freed. Subsequent references will have no effect.
<code>.rn <i>xx yy</i></code>	-	ignored	-	Rename request, macro, or string <i>xx</i> to <i>yy</i> . If <i>yy</i> exists, it is first removed.
<code>.di <i>xx</i></code>	-	end	D	Divert output to macro <i>xx</i> . Normal text processing occurs during diversion except that page offsetting is not done. The diversion ends when the request <code>di</code> or <code>da</code> is encountered without an argument; extraneous requests of this type should not appear when nested diversions are being used.
<code>.da <i>xx</i></code>	-	end	D	Divert, appending to <i>xx</i> (append version of <code>di</code>).
<code>.wh <i>N xx</i></code>	-	-	v	Install a trap to invoke <i>xx</i> at page position <i>N</i> ; a <i>negative N</i> will be interpreted with respect to the page <i>bottom</i> . Any macro previously planted at <i>N</i> is replaced by <i>xx</i> . A zero <i>N</i> refers to the <i>top</i> of a page. In the absence of <i>xx</i> , the first found trap at <i>N</i> , if any, is removed.
<code>.ch <i>xx N</i></code>	-	-	v	Change the trap position for macro <i>xx</i> to be <i>N</i> . In the absence of <i>N</i> , the trap, if any, is removed.
<code>.dt <i>N xx</i></code>	-	off	D,v	Install a diversion trap at position <i>N</i> in the <i>current</i> diversion to invoke macro <i>xx</i> . Another <code>dt</code> will redefine the diversion trap. If no arguments are given, the diversion trap is removed.
<code>.it <i>N xx</i></code>	-	off	E	Set an input-line-count trap to invoke the macro <i>xx</i> after <i>N</i> lines of <i>text</i> input have been read (control or request lines don't count). The text may be in-line text or text interpolated by inline or trap-invoked macros.
<code>.em <i>xx</i></code>	none	none	-	The macro <i>xx</i> will be invoked when all input has ended. The effect is the same as if the contents of <i>xx</i> had been at the end of the last file processed.

8. Number Registers

A variety of parameters are available to the user as predefined, named *number registers* (see Summary and Index, page 7). In addition, the user may define his own named registers. Register names are one or two characters long and *do not* conflict with request, macro, or string names. Except for certain predefined read-only registers, a number register can be read, written, automatically incremented or decremented, and interpolated into the input in a variety of formats. One common use of user-defined registers is to automatically number sections, paragraphs, lines, etc. A number register may be used any time numerical input is expected or desired and may be used in numerical *expressions* (§1.4).

Number registers are created and modified using `nr`, which specifies the name, numerical value, and the auto-increment size. Registers are also modified, if accessed with an auto-incrementing sequence. If the registers *x* and *xx* both contain *N* and have the auto-increment size *M*, the following access sequences have the effect shown:

Tab type	Length of motion or repeated characters	Location of <i>next-string</i>
Left	D	Following D
Right	$D-W$	Right adjusted within D
Centered	$D-W/2$	Centered on right end of D

The length of generated motion is allowed to be negative, but that of a repeated character string cannot be. Repeated character strings contain an integer number of characters, and any residual distance is prepended as motion. Tabs or leaders found after the last tab stop are ignored, but may be used as *next-string* terminators.

Tabs and leaders are not interpreted in *copy mode*. `\t` and `\a` always generate a non-interpreted tab and leader respectively, and are equivalent to actual tabs and leaders in *copy mode*.

9.2. Fields. A *field* is contained between a *pair of field delimiter* characters, and consists of sub-strings separated by *padding* indicator characters. The field length is the distance on the *input* line from the position where the field begins to the next tab stop. The difference between the total length of all the sub-strings and the field length is incorporated as horizontal padding space that is divided among the indicated padding places. The incorporated padding is allowed to be negative. For example, if the field delimiter is `#` and the padding indicator is `^`, `^xxx^right#` specifies a right-adjusted string with the string `xxx` centered in the remaining space.

Request Form	Initial Value	If No Argument	Notes	Explanation
<code>.ta Nt ...</code>	8n; 0.5in	none	E,m	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.
<code>.tc c</code>	none	none	E	The tab repetition character becomes <i>c</i> , or is removed specifying motion.
<code>.lc c</code>	.	none	E	The leader repetition character becomes <i>c</i> , or is removed specifying motion.
<code>.fc a b</code>	off	off	-	The field delimiter is set to <i>a</i> ; the padding indicator is set to the <i>space</i> character or to <i>b</i> , if given. In the absence of arguments the field mechanism is turned off.

10. Input and Output Conventions and Character Translations

10.1. Input character translations. Ways of inputting the graphic character set were discussed in §2.1. The ASCII control characters horizontal tab (§9.1), SOH (§9.1), and backspace (§10.3) are discussed elsewhere. The newline delimits input lines. In addition, STX, ETX, ENQ, ACK, and BEL are accepted, and may be used as delimiters or translated into a graphic with `tr` (§10.5). *All others are ignored.*

The *escape* character `\` introduces *escape sequences*—causes the following character to mean another character, or to indicate some function. A complete list of such sequences is given in the Summary and Index on page 6. `\` should not be confused with the ASCII control character ESC of the same name. The escape character `\` can be input with the sequence `\\`. The escape character can be changed with `ec`, and all that has been said about the default `\` becomes true for the new escape character. `\e` can be used to print whatever the current escape character is. If necessary or convenient, the escape mechanism may be turned off with `eo`, and restored with `ec`.

Request Form	Initial Value	If No Argument	Notes	Explanation
.tr <i>abcd</i> ...	none	-	O	Translate <i>a</i> into <i>b</i> , <i>c</i> into <i>d</i> , etc. If an odd number of characters is given, the last one will be mapped into the space character. To be consistent, a particular translation must stay in effect from <i>input</i> to <i>output</i> time.

10.6. *Transparent throughput.* An input line beginning with a `!` is read in *copy mode* and *transparently* output (without the initial `!`); the text processor is otherwise unaware of the line's presence. This mechanism may be used to pass control information to a post-processor or to imbed control lines in a macro created by a diversion.

10.7. *Comments and concealed newlines.* An uncomfortably long input line that must stay one line (e.g. a string definition, or nofilled text) can be split into many physical lines by ending all but the last one with the escape `\`. The sequence `\(newline)` is *always* ignored—except in a comment. Comments may be imbedded at the *end* of any line by prefacing them with `\`. The newline at the end of a comment cannot be concealed. A line beginning with `\`` will appear as a blank line and behave like `.sp 1`; a comment can be on a line by itself by beginning the line with `.\`.`

11. Local Horizontal and Vertical Motions, and the Width Function

11.1. *Local Motions.* The functions `\v'N'` and `\h'N'` can be used for *local* vertical and horizontal motion respectively. The distance *N* may be negative; the *positive* directions are *rightward* and *downward*. A *local* motion is one contained *within* a line. To avoid unexpected vertical dislocations, it is necessary that the *net* vertical local motion within a word in filled text and otherwise within a line balance to zero. The above and certain other escape sequences providing local motion are summarized in the following table.

Vertical Local Motion	Effect in		Horizontal Local Motion	Effect in	
	TROFF	NROFF		TROFF	NROFF
<code>\v'N'</code>	Move distance <i>N</i>		<code>\h'N'</code> <code>\(space)</code> <code>\0</code>	Move distance <i>N</i> Unpaddable space-size space Digit-size space	
<code>\u</code> <code>\d</code> <code>\r</code>	1/2 em up 1/2 em down 1 em up	1/2 line up 1/2 line down 1 line up	<code>\ </code> <code>\^</code>	1/6 em space 1/12 em space	ignored 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 `\kx` will cause the *current* horizontal position in the *input line* to be stored in register *x*. As an example, the construction `\kxword\h'| \nxu+2u'word` will embolden *word* by backing up to almost its beginning and overprinting it, resulting in *word*.



The horizontal and vertical line drawing functions may be used in combination to produce large boxes. The zero-width *box-rule* and the 1/2-em wide *underrule* were *designed* to form corners when using 1-em vertical spacings. For example the macro

```
.de eb
.sp -1      \"compensate for next automatic base-line spacing
.nf        \"avoid possibly overflowing word buffer
\h'-.5n'\L'|\\nau-1'I'\n(.lu+1n(ul'\L'-|\\nau+1'I'|0u-.5n(ul'  \"draw box
.fi
..
```

will draw a box around some text whose beginning vertical place was saved in number register *a* (e. g. using *.mk a*) as done for this paragraph.

13. Hyphenation.

The automatic hyphenation may be switched off and on. When switched on with *hy*, several variants may be set. A *hyphenation indicator* character may be imbedded in a word to specify desired hyphenation points, or may be prepended to suppress hyphenation. In addition, the user may specify a small exception word list.

Only words that consist of a central alphabetic string surrounded by (usually null) non-alphabetic strings are considered candidates for automatic hyphenation. Words that were input containing hyphens (minus), em-dashes (*\(em)*), or hyphenation indicator characters—such as *mother-in-law*—are *always* subject to splitting after those characters, whether or not automatic hyphenation is on or off.

Request Form	Initial Value	If No Argument	Notes	Explanation
<i>.nh</i>	hyphenate	-	E	Automatic hyphenation is turned off.
<i>.hyN</i>	on, <i>N</i> =1	on, <i>N</i> =1	E	Automatic hyphenation is turned on for <i>N</i> ≥1, or off for <i>N</i> =0. If <i>N</i> =2, <i>last</i> lines (ones that will cause a trap) are not hyphenated. For <i>N</i> =4 and 8, the last and first two characters respectively of a word are not split off. These values are additive; i. e. <i>N</i> =14 will invoke all three restrictions.
<i>.hc c</i>	\%	\%	E	Hyphenation indicator character is set to <i>c</i> or to the default \%. The indicator does not appear in the output.
<i>.hw word1 ...</i>		ignored	-	Specify hyphenation points in words with imbedded minus signs. Versions of a word with terminal <i>s</i> are implied; i. e. <i>dig-it</i> implies <i>dig-its</i> . This list is examined initially <i>and</i> after each suffix stripping. The space available is small—about 128 characters.

14. Three Part Titles.

The titling function *tl* provides for automatic placement of three fields at the left, center, and right of a line with a title-length specifiable with *lt*. *tl* may be used anywhere, and is independent of the normal text collecting process. A common use is in header and footer macros.

Request Form	Initial Value	If No Argument	Notes	Explanation
<i>.tl 'left' 'center' 'right'</i>		-	-	The strings <i>left</i> , <i>center</i> , and <i>right</i> are respectively left-adjusted, centered, and right-adjusted in the current title-length. Any of the strings may be empty, and overlapping is permitted. If the page-number character (initially %) is found within any of the fields it is replaced by the current page number having the format assigned

`.ie c anything` - `u` If portion of if-else; all above forms (like `if`).
`.el anything` - - Else portion of if-else.

The built-in condition names are:

Condition Name	True If
<code>o</code>	Current page number is odd
<code>e</code>	Current page number is even
<code>t</code>	Formatter is TROFF
<code>n</code>	Formatter is NROFF

If the condition *c* is *true*, or if the number *N* is greater than zero, or if the strings compare identically (including motions and character size and font), *anything* is accepted as input. If a `!` precedes the condition, number, or string comparison, the sense of the acceptance is reversed.

Any spaces between the condition and the beginning of *anything* are skipped over. The *anything* can be either a single input line (text, macro, or whatever) or a number of input lines. In the multi-line case, the first line must begin with a left delimiter `\{` and the last line must end with a right delimiter `\}`.

The request `ie` (if-else) is identical to `if` except that the acceptance state is remembered. A subsequent and matching `el` (else) request then uses the reverse sense of that state. `ie - el` pairs may be nested.

Some examples are:

```
.if e .tl ' Even Page %'''
```

which outputs a title if the page number is even; and

```
.ie \n%>1 \{\
.sp 0.5i
.tl ' Page %'''\
.sp | 1.2i \}
.el .sp | 2.5i
```

which treats page 1 differently from other pages.

17. Environment Switching.

A number of the parameters that control the text processing are gathered together into an *environment*, which can be switched by the user. The environment parameters are those associated with requests noting `E` in their *Notes* column; in addition, partially collected lines and words are in the environment. Everything else is global; examples are page-oriented parameters, diversion-oriented parameters, number registers, and macro and string definitions. All environments are initialized with default parameter values.

Request Form	Initial Value	If No Argument	Notes	Explanation
<code>.ev N</code>	$N=0$	previous	-	Environment switched to environment $0 \leq N \leq 2$. Switching is done in push-down fashion so that restoring a previous environment <i>must</i> be done with <code>.ev</code> 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*.

- .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.
- .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 - - B 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.

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.



difference in vertical base-line spacings of the two environments, to prevent the late triggering the footer trap from causing the last line of the combined footnotes to overflow. The footer trap is then set to the lower (on the page) of *y* or the current page position (*nl*) plus one line, to allow for printing the reference line. If indicated by *x*, the footer *fo* rereads the footnotes from *FN* in *nofill* mode in environment 1, and deletes *FN*. If the footnotes were too large to fit, the macro *fx* will be trap-invoked to divert 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 II

Input Naming Conventions for ' , ` , and -
and for Non-ASCII Special Characters

Non-ASCII characters and *minus* on the standard fonts.

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

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.

Input Character			Input Character		
Char	Name	Name	Char	Name	Name
+	\(pl	math plus	κ	\(*k	kappa
-	\(mi	math minus	λ	\(*l	lambda
=	\(eq	math equals	μ	\(*m	mu
*	\(**	math star	ν	\(*n	nu
§	\(sc	section	ξ	\(*c	xi
´	\(aa	acute accent	ο	\(*o	omicron
˘	\(ga	grave accent	π	\(*p	pi
˘	\(ul	underrule	ρ	\(*r	rho
/	\(sl	slash (matching backslash)	σ	\(*s	sigma
α	\(*a	alpha	ς	\(ts	terminal sigma
β	\(*b	beta	τ	\(*t	tau
γ	\(*g	gamma	υ	\(*u	upsilon
δ	\(*d	delta	φ	\(*f	phi
ε	\(*e	epsilon	χ	\(*x	chi
ζ	\(*z	zeta	ψ	\(*q	psi
η	\(*y	eta	ω	\(*w	omega
θ	\(*h	theta	Α	\(*A	Alpha†
ι	\(*i	iota	Β	\(*B	Beta†



A TROFF Tutorial

Brian W. Kernighan
(updated for 4.3BSD by Mark Seiden)

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

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 (as on this page), a full greek alphabet, and a substantial number of special characters and mathematical symbols. Characters can be printed in a range of sizes, and placed anywhere on the page.

troff allows the user full control over fonts, sizes, and character positions, as well as the usual features of a formatter — right-margin justification, automatic hyphenation, page titling and numbering, and so on. It also provides macros, arithmetic variables and operations, and conditional testing, for complicated formatting tasks.

This document is an introduction to the most basic use of **troff**. It presents just enough information to enable the user to do simple formatting tasks like making viewgraphs, and to make incremental changes to existing packages of **troff** commands. In most respects, the UNIX formatter **nroff** 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.

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

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

size, except that \s0 causes the size to revert to its previous value. Notice that \s1011 can be understood correctly as 'size 10, followed by an 11', if the size is legal, but not otherwise. Be cautious with similar constructions.

Relative size changes are also legal and useful:

```
\s-2UNIX\s+2
```

temporarily decreases the size, whatever it is, by two points, then restores it. Relative size changes have the advantage that the size difference is independent of the starting size of the document. The amount of the relative change is restricted to a single digit.

The other parameter that determines what the type looks like is the spacing between lines, which is set independently of the point size. Vertical spacing is measured from the bottom of one line to the bottom of the next. The command to control vertical spacing is .vs. For running text, it is usually best to set the vertical spacing about 20% bigger than the character size. For example, so far in this document, we have used "9 on 11", that is,

```
.ps 9
.vs 11p
```

If we changed to

```
.ps 9
.vs 9p
```

the running text would look like this. After a few lines, you will agree it looks a little cramped. The right vertical spacing is partly a matter of taste, depending on how much text you want to squeeze into a given space, and partly a matter of traditional printing style. By default, troff uses 10 on 12.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. This is 12 on 14.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. For example, 10 on 12 uses about twice as much space as 7 on 8. This is 6 on 7, which is even smaller. It packs a lot more words per line, but you can go blind trying to read it.

When used without arguments, .ps and .vs revert to the previous size and vertical spacing respectively.

The command .sp is used to get extra vertical space. Unadorned, it gives you one extra blank line (one .vs, whatever that has been set to). Typically, that's more or less than you want, so .sp can be followed by information about how much space you want —

```
.sp 2i
```

means 'two inches of vertical space'.

```
.sp 2p
```

means 'two points of vertical space'; and

```
.sp 2
```

means 'two vertical spaces' — two of whatever .vs is set to (this can also be made explicit with .sp 2v); troff also understands decimal fractions in most places, so

```
.sp 1.5i
```

is a space of 1.5 inches. These same scale factors can be used after .vs to define line spacing, and in fact after most commands that deal with physical dimensions.

It should be noted that all size numbers are converted internally to 'machine units', which are 1/432 inch (1/6 point). For most purposes, this is enough resolution that you don't have to worry about the accuracy of the representation. The situation is not quite so good vertically, where resolution is 1/144 inch (1/2 point).

3. Fonts and Special Characters

troff and the typesetter allow four different fonts at any one time. Normally three fonts (Times roman, italic and bold) and one collection of special characters are permanently mounted.

```
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNopQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNopQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNopQRSTUVWXYZ
```

The greek, mathematical symbols and miscellany of the special font are listed in Appendix A.

troff prints in roman unless told otherwise. To switch into bold, use the .ft command

```
.ft B
```

and for italics,

```
.ft I
```

To return to roman, use .ft R; to return to the previous font, whatever it was, use either .ft P or just .ft. The 'underline' command

```
.ul
```

causes the next input line to print in italics. .ul can be followed by a count to indicate that more than one line is to be italicized.

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

```
bold/face text
```

is produced by

```
.in 0.3i
.ll -0.3i
text to be set into a block
.ll +0.3i
.in -0.3i
```

will create a block that looks like this:

```
Pater noster qui est in caelis sanctificetur
nomen tuum; adveniat regnum tuum; fiat
voluntas tua, sicut in caelo, et in terra. ...
Amen.
```

Notice the use of '+' and '-' to specify the amount of change. These change the previous setting by the specified amount, rather than just overriding it. The distinction is quite important: `.ll +1i` makes lines one inch longer; `.ll 1i` makes them one inch *long*.

With `.in`, `.ll` and `.po`, the previous value is used if no argument is specified.

To indent a single line, use the 'temporary indent' command `.ti`. For example, all paragraphs in this memo effectively begin with the command

```
.ti 3
```

Three of what? The default unit for `.ti`, as for most horizontally oriented commands (`.ll`, `.in`, `.po`), is ems; an em is roughly the width of the letter 'm' in the current point size. (Precisely, a em in size *p* is *p* points.) Although inches are usually clearer than ems to people who don't set type for a living, ems have a place: they are a measure of size that is proportional to the current point size. If you want to make text that keeps its proportions regardless of point size, you should use ems for all dimensions. Em's 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).

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 tab  50 tab  60
   700 tab 800 tab  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 straight-forward, but messy to read and tough to type correctly.


```
.wh -li 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 ‘-li’ 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 -li NP
```

Now what happens? As text is actually being output, `troff` keeps track of its vertical position on the page, and after a line is printed within one inch from the bottom, the `.NP` macro is activated. (In the jargon, the `.wh` command sets a *trap* at the specified place, which is ‘sprung’ when that point is passed.) `.NP` causes a skip to the top of the next page (that’s what the ‘bp was for), then prints the title with the appropriate margins.

Why ‘bp and ‘sp instead of ‘.bp and ‘.sp? The answer is that `.sp` and `.bp`, like several other commands, cause a *break* to take place. That is, all the input text collected but not yet printed is flushed out as soon as possible, and the next input line is guaranteed to start a new line of output. If we had used `.sp` or `.bp` in the `.NP` macro, this would cause a break in the middle of the current output line when a new page is started. The effect would be to print the left-over part of that line at the top of the page, followed by the next input line on a new output line. This is *not* what we want. Using ‘ instead of . for a command tells `troff` that no break is to take place — the output line currently being filled should *not* be forced out before the space or new page.

The list of commands that cause a break is short and natural:

```
.bp .br .ce .fi .nf .sp .in .ti
```

All others cause *no* break, regardless of whether you use a . or a ‘. If you really need a break, add a `.br` command at the appropriate place.

One other thing to beware of — if you’re changing fonts or point sizes a lot, you may find that if you cross a page boundary in an unexpected font or size, your titles come out in that size and font instead of what you intended. Furthermore, the length of a title is independent of the current line length, so titles will come out at the default length of 6.5 inches unless you change it, which is done with the `.lt` command.

There are several ways to fix the problems of point sizes and fonts in titles. For the simplest applications, we can change `.NP` to set the proper

size and font for the title, then restore the previous values, like this:

```
.de NP
'bp
'sp 0.5i
.ft R      \" set title font to roman
.ps 10     \" and size to 10 point
.lt 6i     \" and length to 6 inches
.tl left'center'right'
.ps        \" revert to previous size
.ft P     \" and to previous font
'sp 0.3i
..
```

This version of `.NP` does *not* work if the fields in the `.tl` command contain size or font changes. To cope with that requires `troff`’s ‘environment’ mechanism, which we will discuss in Section 13.

To get a footer at the bottom of a page, you can modify `.NP` so it does some processing before the ‘bp command, or split the job into a footer macro invoked at the bottom margin and a header macro invoked at the top of the page. These variations are left as exercises.

Output page numbers are computed automatically as each page is produced (starting at 1), but no numbers are printed unless you ask for them explicitly. To get page numbers printed, include the character % in the `.tl` line at the position where you want the number to appear. For example

```
.tl ~- %-
```

centers the page number inside hyphens, as on this page. You can set the page number at any time with either `.bp n`, which immediately starts a new page numbered *n*, or with `.pn n`, which sets the page number for the next page but doesn’t cause a skip to the new page. Again, `.bp +n` sets the page number to *n* more than its current value; `.bp` means `.bp +1`.

10. Number Registers and Arithmetic

`troff` has a facility for doing arithmetic, and for defining and using variables with numeric values, called *number registers*. Number registers, like strings and macros, can be useful in setting up a document so it is easy to change later. And of course they serve for any sort of arithmetic computation.

Like strings, number registers have one or two character names. They are set by the `.nr` command, and are referenced anywhere by `\nx` (one character name) or `\n(xy)` (two character name).

There are quite a few pre-defined number registers maintained by `troff`, among them % for the current page number; nl for the current vertical position on the page; dy, mo and yr for the current day,

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\fi\\$1\h-\w\\$1u+1u\\$1\fp\\$2
```

The `\h` and `\w` commands need no extra backslash, as we discussed above. The `&` is there in case the argument begins with a period.

Two backslashes are needed with the `\\$n` commands, though, to protect one of them when the macro is being defined. Perhaps a second example will make this clearer. Consider a macro called `.SH` which produces section headings rather like those in this paper, with the sections numbered automatically, and the title in bold in a smaller size. The use is

```
.SH "Section title ..."
```

(If the argument to a macro is to contain blanks, then it must be *surrounded* by double quotes, unlike a string, where only one leading quote is permitted.)

Here is the definition of the `.SH` macro:

```
.nr SH 0 \ " initialize section number
.de SH
.sp 0.3i
.ft B
.nr SH \\n(SH+1 \ " increment number
.ps \\n(PS-1\ " decrease PS
\\n(SH. \\$1 \ " number, title
.ps \\n(PS \ " restore PS
.sp 0.3i
.ft R
..
```

The section number is kept in number register `SH`, which is incremented each time just before it is used. (A number register may have the same name as a macro without conflict but a string may not.)

We used `\\n(SH` instead of `\\n(SH` and `\\n(PS` instead of `\\n(PS`. If we had used `\\n(SH`, we would get the value of the register at the time the macro was *defined*, not at the time it was *used*. If that's what you want, fine, but not here. Similarly, by using `\\n(PS`, we get the point size at the time the macro is called.

As an example that does not involve numbers, recall our `.NP` macro which had a

```
.tl left'center'right'
```

We could make these into parameters by using instead

```
.tl \\*(LT\\*(CT\\*(RT'
```

so the title comes from three strings called `LT`, `CT` and `RT`. If these are empty, then the title will be a blank line. Normally `CT` would be set with something like

```
.ds CT - % -
```

to give just the page number between hyphens (as on the top of this page), but a user could supply private definitions for any of the strings.

12. Conditionals

Suppose we want the `.SH` macro to leave two extra inches of space just before section 1, but nowhere else. The cleanest way to do that is to test inside the `.SH` macro whether the section number is 1, and add some space if it is. The `.if` command provides the conditional test that we can add just before the heading line is output:

```
.if \\n(SH=1 .sp 2i \ " first section only
```

The condition after the `.if` can be any arithmetic or logical expression. If the condition is logically true, or arithmetically greater than zero, the rest of the line is treated as if it were text — here a command. If the condition is false, or zero or nega-

.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.



A System for Typesetting Mathematics

Brian W. Kernighan and Lorinda L. Cherry

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes the design and implementation of a system for typesetting mathematics. The language has been designed to be easy to learn and to use by people (for example, secretaries and mathematical typists) who know neither mathematics nor typesetting. Experience indicates that the language can be learned in an hour or so, for it has few rules and fewer exceptions. For typical expressions, the size and font changes, positioning, line drawing, and the like necessary to print according to mathematical conventions are all done automatically. For example, the input

sum from i=0 to infinity x sub i = pi over 2

produces

$$\sum_{i=0}^{\infty} x_i = \frac{\pi}{2}$$

The syntax of the language is specified by a small context-free grammar; a compiler-compiler is used to make a compiler that translates this language into typesetting commands. Output may be produced on either a phototypesetter or on a terminal with forward and reverse half-line motions. The system interfaces directly with text formatting programs, so mixtures of text and mathematics may be handled simply.

This paper is a revision of a paper originally published in CACM, March, 1975.

1. Introduction

"Mathematics is known in the trade as *difficult*, or *penalty copy* because it is slower, more difficult, and more expensive to set in type than any other kind of copy normally occurring in books and journals." [1]

One difficulty with mathematical text is the multiplicity of characters, sizes, and fonts. An expression such as

$$\lim_{x \rightarrow \pi/2} (\tan x)^{\sin 2x} = 1$$

requires an intimate mixture of roman, italic and greek letters, in three sizes, and a special character or two. ("Requires" is perhaps the wrong word, but mathematics has its own typographical conventions which are quite different from those of ordinary text.) Typesetting such an expression by traditional methods is still an essentially manual operation.

A second difficulty is the two dimensional character of mathematics, which the superscript and

limits in the preceding example showed in its simplest form. This is carried further by

$$a_0 + \frac{b_1}{a_1 + \frac{b_2}{a_2 + \frac{b_3}{a_3 + \dots}}}$$

and still further by

$$\int \frac{dx}{ae^{mx} - be^{-mx}} = \begin{cases} \frac{1}{2m\sqrt{ab}} \log \frac{\sqrt{a}e^{mx} - \sqrt{b}}{\sqrt{a}e^{mx} + \sqrt{b}} \\ \frac{1}{m\sqrt{ab}} \tanh^{-1} \left(\frac{\sqrt{a}}{\sqrt{b}} e^{mx} \right) \\ \frac{-1}{m\sqrt{ab}} \coth^{-1} \left(\frac{\sqrt{a}}{\sqrt{b}} e^{mx} \right) \end{cases}$$

These examples also show line-drawing, built-up characters like braces and radicals, and a spectrum of positioning problems. (Section 6 shows what a user has to type to produce these on our system.)



text, but marked by user settable delimiters. The program reads this input and treats as comments those things which are not mathematics, simply passing them through untouched. At the same time it converts the mathematical input into the necessary TROFF commands. The resulting ioutput is passed directly to TROFF where the comments and the mathematical parts both become text and/or TROFF commands.

4. The Language

We will not try to describe the language precisely here; interested readers may refer to the appendix for more details. Throughout this section, we will write expressions exactly as they are handed to the typesetting program (hereinafter called "EQN"), except that we won't show the delimiters that the user types to mark the beginning and end of the expression. The interface between EQN and TROFF is described at the end of this section.

As we said, typing $x=y+z+1$ should produce $x=y+z+1$, and indeed it does. Variables are made italic, operators and digits become roman, and normal spacings between letters and operators are altered slightly to give a more pleasing appearance.

Input is free-form. Spaces and new lines in the input are used by EQN to separate pieces of the input; they are not used to create space in the output. Thus

$$x = y + z + 1$$

also gives $x=y+z+1$. Free-form input is easier to type initially; subsequent editing is also easier, for an expression may be typed as many short lines.

Extra white space can be forced into the output by several characters of various sizes. A tilde "~" gives a space equal to the normal word spacing in text; a circumflex gives half this much, and a tab charcter spaces to the next tab stop.

Spaces (or tildes, etc.) also serve to delimit pieces of the input. For example, to get

$$f(t)=2\pi\int\sin(\omega t)dt$$

we write

$$f(t) = 2 \text{ pi int sin (omega t) dt}$$

Here spaces are *necessary* in the input to indicate that *sin*, *pi*, *int*, and *omega* are special, and potentially worth special treatment. EQN looks up each such string of characters in a table, and if appropriate gives it a translation. In this case, *pi* and *omega* become their greek equivalents, *int* becomes the integral sign (which must be moved down and enlarged so it looks "right"), and *sin* is made roman, following conventional mathematical practice. Parentheses, digits and operators are automatically

made roman wherever found.

Fractions are specified with the keyword *over*:

$$a+b \text{ over } c+d+e = 1$$

produces

$$\frac{a+b}{c+d+e} = 1$$

Similarly, subscripts and superscripts are introduced by the keywords *sub* and *sup*:

$$x^2+y^2=z^2$$

is produced by

$$x \text{ sup } 2 + y \text{ sup } 2 = z \text{ sup } 2$$

The spaces after the 2's are necessary to mark the end of the superscripts; similarly the keyword *sup* has to be marked off by spaces or some equivalent delimiter. The return to the proper baseline is automatic. Multiple levels of subscripts or superscripts are of course allowed: "x sup y sup z" is x^{y^z} . The construct "something *sub* something *sup* something" is recognized as a special case, so "x sub i sup 2" is x_i^2 instead of x_i^2 .

More complicated expressions can now be formed with these primitives:

$$\frac{\partial^2 f}{\partial x^2} = \frac{x^2}{a^2} + \frac{y^2}{b^2}$$

is produced by

$$(\text{partial sup } 2 \text{ f}) \text{ over } (\text{partial } x \text{ sup } 2) = x \text{ sup } 2 \text{ over } a \text{ sup } 2 + y \text{ sup } 2 \text{ over } b \text{ sup } 2$$

Braces {} are used to group objects together; in this case they indicate unambiguously what goes over what on the left-hand side of the expression. The language defines the precedence of *sup* to be higher than that of *over*, so no braces are needed to get the correct association on the right side. Braces can always be used when in doubt about precedence.

The braces convention is an example of the power of using a recursive grammar to define the language. It is part of the language that if a construct can appear in some context, then *any expression* in braces can also occur in that context.

There is a *sqr*t operator for making square roots of the appropriate size: "sqr a+b" produces $\sqrt{a+b}$, and

$$x = \{-b + \text{sqr}(b \text{ sup } 2 - 4ac)\} \text{ over } 2a$$

is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Since large radicals look poor on our typesetter, *sqr*t is not useful for tall expressions.

input:

```
.ce
.EQ
x sub i = y sub i ...
.EN
```

Since it is tedious to type “.EQ” and “.EN” around very short expressions (single letters, for instance), the user can also define two characters to serve as the left and right delimiters of expressions. These characters are recognized anywhere in subsequent text. For example if the left and right delimiters have both been set to “#”, the input:

Let #x sub i#, #y# and #alpha# be positive

produces:

Let x_i , y and α be positive

Running a preprocessor is strikingly easy on UNIX. To typeset text stored in file “f”, one issues the command:

eqn f | troff

The vertical bar connects the output of one process (EQN) to the input of another (TROFF).

5. Language Theory

The basic structure of the language is not a particularly original one. Equations are pictured as a set of “boxes,” pieced together in various ways. For example, something with a subscript is just a box followed by another box moved downward and shrunk by an appropriate amount. A fraction is just a box centered above another box, at the right altitude, with a line of correct length drawn between them.

The grammar for the language is shown below. For purposes of exposition, we have collapsed some productions. In the original grammar, there are about 70 productions, but many of these are simple ones used only to guarantee that some keyword is recognized early enough in the parsing process. Symbols in capital letters are terminal symbols; lower case symbols are non-terminals, i.e., syntactic categories. The vertical bar | indicates an alternative; the brackets [] indicate optional material. A TEXT is a string of non-blank characters or any string inside double quotes; the other terminal symbols represent literal occurrences of the corresponding keyword.

```
eqn : box | eqn box
box : text
    | { eqn }
    | box OVER box
    | SQRT box
    | box SUB box | box SUP box
    | [ L | C | R ]PILE { list }
    | LEFT text eqn [ RIGHT text ]
    | box [ FROM box ] [ TO box ]
    | SIZE text box
    | [ ROMAN | BOLD | ITALIC ] box
    | box [ HAT | BAR | DOT | DOTDOT | TILDE ]
    | DEFINE text text
list : eqn | list ABOVE eqn
text : TEXT
```

The grammar makes it obvious why there are few exceptions. For example, the observation that something can be replaced by a more complicated something in braces is implicit in the productions:

```
eqn : box | eqn box
box : text | { eqn }
```

Anywhere a single character could be used, any legal construction can be used.

Clearly, our grammar is highly ambiguous. What, for instance, do we do with the input

a over b over c ?

Is it

(a over b) over c

or is it

a over (b over c) ?

To answer questions like this, the grammar is supplemented with a small set of rules that describe the precedence and associativity of operators. In particular, we specify (more or less arbitrarily) that *over* associates to the left, so the first alternative above is the one chosen. On the other hand, *sub* and *sup* bind to the right, because this is closer to standard mathematical practice. That is, we assume x^{a^b} is $x^{(a^b)}$, not $(x^a)^b$.

The precedence rules resolve the ambiguity in a construction like

a sup 2 over b

We define *sup* to have a higher precedence than *over*, so this construction is parsed as $\frac{a^2}{b}$ instead of $a \frac{2}{b}$.

Naturally, a user can always force a particular parsing by placing braces around expressions.

The ambiguous grammar approach seems to be quite useful. The grammar we use is small enough to be easily understood, for it contains none



interesting than a regular typewriter.

The main difficulty that users have seems to be remembering that a blank is a delimiter; even experienced users use blanks where they shouldn't and omit them when they are needed. A common instance is typing

$$f(x \text{ sub } i)$$

which produces

$$f(x_i)$$

instead of

$$f(x_i)$$

Since the EQN language knows no mathematics, it cannot deduce that the right parenthesis is not part of the subscript.

The language is somewhat prolix, but this doesn't seem excessive considering how much is being done, and it is certainly more compact than the corresponding TROFF commands. For example, here is the source for the continued fraction expression in Section 1 of this paper:

```
a sub 0 + b sub 1 over
(a sub 1 + b sub 2 over
(a sub 2 + b sub 3 over
(a sub 3 + ... )))
```

This is the input for the large integral of Section 1; notice the use of definitions:

```
define emx "(e sup mx)"
define mab "(m sqrt ab)"
define sa "(sqrt a)"
define sb "(sqrt b)"
int dx over {a emx - be sup -mx} ~ ~
left { lpile {
  l over {2 mab} ~ log ~
  (sa emx - sb) over (sa emx + sb)
above
  l over mab ~ tanh sup -1 (sa over sb emx)
above
  -1 over mab ~ coth sup -1 (sa over sb emx)
}
```

As to ease of construction, we have already mentioned that there are really only a few person-months invested. Much of this time has gone into two things—fine-tuning (what is the most esthetically pleasing space to use between the numerator and denominator of a fraction?), and changing things found deficient by our users (shouldn't a tilde be a delimiter?).

The program consists of a number of small, essentially unconnected modules for code generation, a simple lexical analyzer, a canned parser which we did not have to write, and some miscellany associated with input files and the macro facility. The program is now about 1600 lines of C [6], a

high-level language reminiscent of BCPL. About 20 percent of these lines are "print" statements, generating the output code.

The semantic routines that generate the actual TROFF commands can be changed to accommodate other formatting languages and devices. For example, in less than 24 hours, one of us changed the entire semantic package to drive NROFF, a variant of TROFF, for typesetting mathematics on teletypewriter devices capable of reverse line motions. Since many potential users do not have access to a typesetter, but still have to type mathematics, this provides a way to get a typed version of the final output which is close enough for debugging purposes, and sometimes even for ultimate use.

7. Conclusions

We think we have shown that it is possible to do acceptably good typesetting of mathematics on a phototypesetter, with an input language that is easy to learn and use and that satisfies many users' demands. Such a package can be implemented in short order, given a compiler-compiler and a decent typesetting program underneath.

Defining a language, and building a compiler for it with a compiler-compiler seems like the only sensible way to do business. Our experience with the use of a grammar and a compiler-compiler has been uniformly favorable. If we had written everything into code directly, we would have been locked into our original design. Furthermore, we would have never been sure where the exceptions and special cases were. But because we have a grammar, we can change our minds readily and still be reasonably sure that if a construction works in one place it will work everywhere.

Acknowledgements

We are deeply indebted to J. F. Ossanna, the author of TROFF, for his willingness to modify TROFF to make our task easier and for his continuous assistance during the development of our program. We are also grateful to A. V. Aho for help with language theory, to S. C. Johnson for aid with the compiler-compiler, and to our early users A. V. Aho, S. I. Feldman, S. C. Johnson, R. W. Hamming, and M. D. McIlroy for their constructive criticisms.

References

- [1] *A Manual of Style*, 12th Edition. University of Chicago Press, 1969. p 295.
- [2] *Model CIA/T Phototypesetter*. Graphic Systems, Inc., Hudson, N. H.
- [3] Ritchie, D. M., and Thompson, K. L., "The UNIX time-sharing system." *Comm. ACM* 17, 7 (July 1974), 365-375 (reprinted here as PS2:1).

Typesetting Mathematics — User's Guide (Second Edition)

Brian W. Kernighan and Lorinda L. Cherry

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This is the user's guide for a system for typesetting mathematics, using the phototypesetters on the UNIX† operating system.

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 $\lim_{x \rightarrow \pi/2} (\tan x)^{\sin 2x} = 1$ or display equations like

$$\begin{aligned} G(z) &= e^{\ln G(z)} = \exp \left[\sum_{k \geq 1} \frac{S_k z^k}{k} \right] = \prod_{k \geq 1} e^{S_k z^k / k} \\ &= \left[1 + S_1 z + \frac{S_1^2 z^2}{2!} + \dots \right] \left[1 + \frac{S_2 z^2}{2} + \frac{S_2^2 z^4}{2^2 \cdot 2!} + \dots \right] \dots \\ &= \sum_{m \geq 0} \left[\sum_{\substack{k_1, k_2, \dots, k_m \geq 0 \\ k_1 + 2k_2 + \dots + mk_m = m}} \frac{S_1^{k_1}}{1^{k_1} k_1!} \frac{S_2^{k_2}}{2^{k_2} k_2!} \dots \frac{S_m^{k_m}}{m^{k_m} k_m!} \right] z^m \end{aligned}$$

can be learned in an hour or so.

The language interfaces directly with the phototypesetting language TROFF, so mathematical expressions can be embedded in the running text of a manuscript, and the entire document produced in one process. This user's guide is an example of its output.

The same language may be used with the UNIX formatter NROFF to set mathematical expressions on DASI and GSI terminals and Model 37 teletypes.

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.

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

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 $\backslash bs$ 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:

$$x \sim 2 \pi \int \sin(\omega t) dt$$

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*.

$$x \text{ sup } 2 + y \text{ sub } k$$

gives

$$x^2 + y_k$$

EQN takes care of all the size changes and vertical motions needed to make the output look right. The words *sub* and *sup* must be surrounded by spaces; $x \text{ sub } 2$ will give you $x_{\text{sub}2}$ 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

$$y = (x \text{ sup } 2) + 1$$

which causes

$$y = (x^2) + 1$$

instead of the intended

$$y = (x^2) + 1$$

Subscripted subscripts and superscripted superscripts also work:

$$x \text{ sub } i \text{ sub } 1$$

is

$$x_{i_1}$$

A subscript and superscript on the same thing are printed one above the other if the subscript comes *first*:

$$x \text{ sub } i \text{ sup } 2$$

is

$$x_i^2$$

Other than this special case, *sub* and *sup* group to the right, so $x \text{ sup } y \text{ sub } z$ 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:

$$e \text{ sup } \{ i \omega t \}$$

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:

$$x \text{ sub } \{ i \text{ sub } 1 \} \text{ sup } 2$$

is

$$x_{i_1}^2$$

with braces, but

$$x \text{ sub } i \text{ sub } 1 \text{ sup } 2$$

is

$$x_{i_1}^?$$

which is rather different.

Braces can occur within braces if necessary:



n.

12. Size and Font Changes

By default, equations are set in 10-point type (the same size as this guide), with standard mathematical conventions to determine what characters are in roman and what in italic. Although EQN makes a valiant attempt to use esthetically pleasing sizes and fonts, it is not perfect. To change sizes and fonts, use *size n* and *roman*, *italic*, *bold* and *fat*. Like *sub* and *sup*, size and font changes affect only the thing that follows them, and revert to the normal situation at the end of it. Thus

bold x y

is

$$xy$$

and

size 14 bold $x = y +$
size 14 ($\alpha + \beta$)

gives

$$x=y+\alpha+\beta$$

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 throughout a document: the global font and size can be changed as often as needed. For example, in a footnote‡ you will typically want the size of equations to match the size of the footnote text, which is two points smaller than the main text. Don't forget to reset the global size at the end of the footnote.

13. Diacritical Marks

To get funny marks on top of letters, there are several words:

x dot	\dot{x}
x dotdot	\ddot{x}
x hat	\hat{x}
x tilde	\tilde{x}
x vec	\vec{x}
x dyad	\overleftrightarrow{x}
x bar	\bar{x}
x under	\underline{x}

The diacritical mark is placed at the right height. The *bar* and *under* are made the right length for the entire construct, as in $\bar{x+y+z}$; 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

‡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*.



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

$$A = \begin{array}{c} a \ x \\ b \ y \\ c \ z \end{array}$$

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 key-word *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

$$\begin{array}{cc} x_i & x^2 \\ y_i & y^2 \end{array}$$

you have to type

```
matrix {
  ccol { x sub i above y sub i }
  ccol { x sup 2 above y sup 2 }
}
```

This produces a matrix with two centered columns. The elements of the columns are then listed just as for a pile, each element separated by the word *above*. You can also use *lcol* or *rcol* to left or right adjust columns. Each column can be separately adjusted, and there can be as many columns as you like.

The reason for using a matrix instead of two adjacent piles, by the way, is that if the elements of the piles don't all have the same height, they won't line up properly. A matrix forces them to line up, because it looks at the entire structure before deciding what spacing to use.

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



22. A Large Example

Here is the complete source for the three display equations in the abstract of this guide.

```
.EQ I
G(z)~mark =~ e sup ( ln ~ G(z) )
~=" exp left (
sum from k>=1 (S sub k z sup k) over k right )
~=" prod from k>=1 e sup (S sub k z sup k /k)
.EN
.EQ I
lineup = left ( 1 + S sub 1 z +
( S sub 1 sup 2 z sup 2 ) over 2! + ... right )
left ( 1+ ( S sub 2 z sup 2 ) over 2
+ ( S sub 2 sup 2 z sup 4 ) over ( 2 sup 2 cdot 2! )
+ ... right ) ...
.EN
.EQ I
lineup = sum from m>=0 left (
sum from
pile ( k sub 1 ,k sub 2 ,..., k sub m >=0
above
k sub 1 +2k sub 2 + ... +mk sub m =m)
( S sub 1 sup (k sub 1) ) over (1 sup k sub 1 k sub 1 ! ) ~
( S sub 2 sup (k sub 2) ) over (2 sup k sub 2 k sub 2 ! ) ~
...
( S sub m sup (k sub m) ) over (m sup k sub m k sub m ! )
right ) z sup m
.EN
```

+ -	±
->	→
<-	←
<<	⇐
>>	⇒
inf	∞
partial	∂
half	½
prime	'
approx	≈
nothing	
cdot	·
times	×
del	∇
grad	∇
...	...
...,	∑
sum	∑
int	∫
prod	∏
union	∪
inter	∩

To obtain Greek letters, simply spell them out in whatever case you want:

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.

>=	≥
<=	≤
= =	≡
!=	≠

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



cisms.

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.

Tbl — A Program to Format Tables

M. E. Lesk

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Tbl is a document formatting preprocessor for *troff* or *nroff* which makes even fairly complex tables easy to specify and enter. It is available on the 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

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 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

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



- 2) **FORMAT.** The format section of the table specifies the layout of the columns. Each line in this section corresponds to one line of the table (except that the last line corresponds to all following lines up to the next .T&, if any — see below), and each line contains a key-letter for each column of the table. It is good practice to separate the key letters for each column by spaces or tabs. Each key-letter is one of the following:

- L** or **l** to indicate a left-adjusted column entry;
- R** or **r** to indicate a right-adjusted column entry;
- C** or **c** to indicate a centered column entry;
- N** or **n** to indicate a numerical column entry, to be aligned with other numerical entries so that the units digits of numbers line up;
- A** or **a** to indicate an alphabetic subcolumn; all corresponding entries are aligned on the left, and positioned so that the widest is centered within the column (see example on page 12);
- S** or **i** 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 right-most dot (.) adjacent to a digit is used as a decimal point; if there is no dot adjoining a digit, the rightmost digit is used as a units digit; if no alignment is indicated, the item is centered in the column. However, the special non-printing character string \& may be used to override unconditionally dots and digits, or to align alphabetic data; this string lines up where a dot normally would, and then disappears from the final output. In the example below, the items shown at the left will be aligned (in a numerical column) as shown on the right:

13	13
4.2	4.2
26.4.12	26.4.12
abc	abc
abc\&	abc
43\&3.22	433.22
749.12	749.12

Note: If numerical data are used in the same column with wider **L** or **r** type table entries, the widest *number* is centered relative to the wider **L** or **r** items (**L** is used instead of **l** for readability; they have the same meaning as key-letters). Alignment within the numerical items is preserved. This is similar to the behavior of a type data, as explained above. However, alphabetic subcolumns (requested by the a key-letter) are always slightly indented relative to **L** items; if necessary, the column width is increased to force this. This is not true for **n** type entries.

Warning: the **n** and **a** items should not be used in the same column.

For readability, the key-letters describing each column should be separated by spaces. The end of the format section is indicated by a period. The layout of the key-letters in the format section resembles the layout of the actual data in the table. Thus a simple format might appear as:

```

c s s
l n n .

```

which specifies a table of three columns. The first line of the table contains a heading centered across all three columns; each remaining line contains a left-adjusted item in the first column followed by two columns of numerical data. A sample table in this format might be:



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(2.5)fi 6
```

Alternative notation

— Instead of listing the format of successive lines of a table on consecutive lines of the format section, successive line formats may be given on the same line, separated by commas, so that the format for the example above might have been written:

```
c s s, l n n .
```

Default

— Column descriptors missing from the end of a format line are assumed to be *L*. The longest line in the format section, however, defines the number of columns in the table; extra columns in the data are ignored silently.

- 3) **DATA.** The data for the table are typed after the format. Normally, each table line is typed as one line of data. Very long input lines can be broken: any line whose last character is ** is combined with the following line (and the ** vanishes). The data for different columns (the table entries) are separated by tabs, or by whatever character has been specified in the option *tabs* option. There are a few special cases:

Troff commands within tables

— An input line beginning with a *‘.* followed by anything but a number is assumed to be a command to *troff* and is passed through unchanged, retaining its position in the table. So, for example, space within a table may be produced by *“.sp”* commands in the data.

Full width horizontal lines

— An input *line* containing only the character *_* (underscore) or *=* (equal sign) is taken to be a single or double line, respectively, extending the full width of the *table*.

Single column horizontal lines

— An input table *entry* containing only the character *_* or *=* is taken to be a single or double line extending the full width of the *column*. Such lines are extended to meet horizontal or vertical lines adjoining this column. To obtain these characters explicitly in a column, either precede them by *&* or follow them by a space before the usual tab or newline.

Short horizontal lines

— An input table *entry* containing only the string *_* is taken to be a single line as wide as the contents of the column. It is not extended to meet adjoining lines.

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 *^*.



the global options such as *box*, or the selection of columns to be made equal width.

Usage.

On UNIX, *tbl* can be run on a simple table with the command

```
tbl input-file | troff
```

but for more complicated use, where there are several input files, and they contain equations and *ms* memorandum layout commands as well as tables, the normal command would be

```
tbl file-1 file-2 . . . | eqn | troff -ms
```

and, of course, the usual options may be used on the *troff* and *eqn* commands. The usage for *nroff* is similar to that for *troff*, but only TELETYPE® Model 37 and Diablo-mechanism (DASI or GSI) terminals can print boxed tables directly.

For the convenience of users employing line printers without adequate driving tables or post-filters, there is a special *-TX* command line option to *tbl* which produces output that does not have fractional line motions in it. The only other command line options recognized by *tbl* are *-ms* and *-mm* which are turned into commands to fetch the corresponding macro files; usually it is more convenient to place these arguments on the *troff* part of the command line, but they are accepted by *tbl* as well.

Note that when *eqn* and *tbl* are used together on the same file *tbl* should be used first. If there are no equations within tables, either order works, but it is usually faster to run *tbl* first, since *eqn* normally produces a larger expansion of the input than *tbl*. However, if there are equations within tables (using the *delim* mechanism in *eqn*), *tbl* must be first or the output will be scrambled. Users must also beware of using equations in *n*-style columns; this is nearly always wrong, since *tbl* attempts to split numerical format items into two parts and this is not possible with equations. The user can defend against this by giving the *delim(xx)* table option; this prevents splitting of numerical columns within the delimiters. For example, if the *eqn* delimiters are \$\$, giving *delim(\$\$)* a numerical column such as "1245 \$+- 16\$" will be divided after 1245, not after 16.

Tbl limits tables to twenty columns; however, use of more than 16 numerical columns may fail because of limits in *troff*, producing the 'too many number registers' message. *Troff* number registers used by *tbl* must be avoided by the user within tables; these include two-digit names from 31 to 99, and names of the forms #*x*, *x*+, *x* |, ~*x*, and *x*-, where *x* is any lower case letter. The names ##, #-, and #^ are also used in certain circumstances. To conserve number register names, the *n* and *a* formats share a register; hence the restriction above that they may not be used in the same column.

For aid in writing layout macros, *tbl* defines a number register TW which is the table width; it is defined by the time that the ".TE" macro is invoked and may be used in the expansion of that macro. More importantly, to assist in laying out multi-page boxed tables the macro T# is defined to produce the bottom lines and side lines of a boxed table, and then invoked at its end. By use of this macro in the page footer a multi-page table can be boxed. In particular, the *ms* macros can be used to print a multi-page boxed table with a repeated heading by giving the argument H to the ".TS" macro. If the table start macro is written

```
.TS H
```

a line of the form

```
.TH
```

must be given in the table after any table heading (or at the start if none). Material up to the ".TH" is placed at the top of each page of table; the remaining lines in the table are placed on several pages as required. Note that this is *not* a feature of *tbl*, but of the *ms* layout macros.

Examples.

Here are some examples illustrating features of *tbl*. The symbol ⊕ in the input represents a tab character.

Input:

```
.TS
box;
c s s
c | c | c
l | l | n.
Major New York Bridges
=
Bridge ⊕ Designer ⊕ Length

̄ Brooklyn ⊕ J. A. Roebling ⊕ 1595
Manhattan ⊕ G. Lindenthal ⊕ 1470
Williamsburg ⊕ L. L. Buck ⊕ 1600

̄ Queensborough ⊕ Palmer & ⊕ 1182
⊕ Hornbostel

̄ ⊕ ⊕ 1380
Triborough ⊕ O. H. Ammann ⊕ _
⊕ ⊕ 383

̄ Bronx Whitestone ⊕ O. H. Ammann ⊕ 2300
Throgs Neck ⊕ O. H. Ammann ⊕ 1800

̄ George Washington ⊕ O. H. Ammann ⊕ 3500
.TE
```

Output:

Major New York Bridges		
Bridge	Designer	Length
Brooklyn	J. A. Roebling	1595
Manhattan	G. Lindenthal	1470
Williamsburg	L. L. Buck	1600
Queensborough	Palmer & Hornbostel	1182
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
c c
np-2 | n | .
⊕ Stack
⊕ _
1 ⊕ 46
⊕ _
2 ⊕ 23
⊕ _
3 ⊕ 15
⊕ _
4 ⊕ 6.5
⊕ _
5 ⊕ 2.1
⊕ _
.TE
```

Output:

	Stack
1	46
2	23
3	15
4	6.5
5	2.1



Input:

.TS
 allbox;
 cfl s s
 c cw(li) cw(li)
 lp9 lp9 lp9.
 New York Area Rocks
 Era ⊕ Formation ⊕ Age (years)
 Precambrian ⊕ Reading Prong ⊕ >1 billion
 Paleozoic ⊕ Manhattan Prong ⊕ 400 million
 Mesozoic ⊕ T(
 .na
 Newark Basin, incl.
 Stockton, Lockatong, and Brunswick
 formations; also Watchungs
 and Palisades.
 T) ⊕ 200 million
 Cenozoic ⊕ Coastal Plain ⊕ T(
 On Long Island 30,000 years;
 Cretaceous sediments redeposited
 by recent glaciation.
 .ad
 T)
 .TE

Output:

<i>New York Area Rocks</i>		
Era	Formation	Age (years)
Precambrian	Reading Prong	>1 billion
Paleozoic	Manhattan Prong	400 million
Mesozoic	Newark Basin, incl. Stockton, Lockatong, and Brunswick formations; also Watchungs and Palisades.	200 million
Cenozoic	Coastal Plain	On Long Island 30,000 years; Cretaceous sediments redeposited by recent glaciation.

Input:

.EQ
 delim \$\$
 .EN
 . . .
 .TS
 doublebox;
 c c
 ll.
 Name ⊕ Definition
 .sp
 .vs +2p
 Gamma ⊕ \$GAMMA (z) = int sub 0 sup inf t sup (z-1) e sup -t dt\$
 Sine ⊕ \$sin (x) = 1 over 2i (e sup ix - e sup -ix)\$
 Error ⊕ \$ roman erf (z) = 2 over sqrt pi int sub 0 sup z e sup {-t sup 2) dt\$
 Bessel ⊕ \$ J sub 0 (z) = 1 over pi int sub 0 sup pi cos (z sin theta) d theta \$
 Zeta ⊕ \$ zeta (s) = sum from k=1 to inf k sup -s ~ (Re s > 1)\$
 .vs -2p
 .TE

Output:

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



Input:

.TS
 c s
 cip-2 s
 l n
 a n.
 Some London Transport Statistics
 (Year 1964)
 Railway route miles ⊙ 244
 Tube ⊙ 66
 Sub-surface ⊙ 22
 Surface ⊙ 156
 .sp .5
 .T&
 l r
 a r.
 Passenger traffic \- railway
 Journeys ⊙ 674 million
 Average length ⊙ 4.55 miles
 Passenger miles ⊙ 3,066 million
 .T&
 l r
 a r.
 Passenger traffic \- road
 Journeys ⊙ 2,252 million
 Average length ⊙ 2.26 miles
 Passenger miles ⊙ 5,094 million
 .T&
 l n
 a n.
 .sp .5
 Vehicles ⊙ 12,521
 Railway motor cars ⊙ 2,905
 Railway trailer cars ⊙ 1,269
 Total railway ⊙ 4,174
 Omnibuses ⊙ 8,347
 .T&
 l n
 a n.
 .sp .5
 Staff ⊙ 73,739
 Administrative, etc. ⊙ 5,582
 Civil engineering ⊙ 5,134
 Electrical eng. ⊙ 1,714
 Mech. eng. \- railway ⊙ 4,310
 Mech. eng. \- road ⊙ 9,152
 Railway operations ⊙ 8,930
 Road operations ⊙ 35,946
 Other ⊙ 2,971
 .TE

Output:

Some London Transport Statistics
 (Year 1964)

Railway route miles	244
Tube	66
Sub-surface	22
Surface	156
Passenger traffic - railway	
Journeys	674 million
Average length	4.55 miles
Passenger miles	3,066 million
Passenger traffic - road	
Journeys	2,252 million
Average length	2.26 miles
Passenger miles	5,094 million
Vehicles	12,521
Railway motor cars	2,905
Railway trailer cars	1,269
Total railway	4,174
Omnibuses	8,347
Staff	73,739
Administrative, etc.	5,582
Civil engineering	5,134
Electrical eng.	1,714
Mech. eng. - railway	4,310
Mech. eng. - road	9,152
Railway operations	8,930
Road operations	35,946
Other	2,971



Output:

New Jersey Representatives (Democrats)		
Name	Office address	Phone
James J. Florio	23 S. White Horse Pike, Somerdale 08083	609-627-8222
William J. Hughes	2920 Atlantic Ave., Atlantic City 08401	609-345-4844
James J. Howard	801 Bangs Ave., Asbury Park 07712	201-774-1600
Frank Thompson, Jr.	10 Rutgers Pl., Trenton 08618	609-599-1619
Andrew Maguire	115 W. Passaic St., Rochelle Park 07662	201-843-0240
Robert A. Roe	U.S.P.O., 194 Ward St., Paterson 07510	201-523-5152
Henry Helstoski	666 Paterson Ave., East Rutherford 07073	201-939-9090
Peter W. Rodino, Jr.	Suite 1435A, 970 Broad St., Newark 07102	201-645-3213
Joseph G. Minish	308 Main St., Orange 07050	201-645-6363
Helen S. Meyner	32 Bridge St., Lambertville 08530	609-397-1830
Dominick V. Daniels	895 Bergen Ave., Jersey City 07306	201-659-7700
Edward J. Patten	Natl. Bank Bldg., Perth Amboy 08861	201-826-4610
(Republicans)		
Millicent Fenwick	41 N. Bridge St., Somerville 08876	201-722-8200
Edwin B. Forsythe	301 Mill St., Moorestown 08057	609-235-6622
Matthew J. Rinaldo	1961 Morris Ave., Union 07083	201-687-4235

This is a paragraph of normal text placed here only to indicate where the left and right margins are. In this way the reader can judge the appearance of centered tables or expanded tables, and observe how such tables are formatted.



Input:

.TS
 expand;
 c s s s
 c c c c
 l l n n.

Bell Labs Locations

Name ⊕ Address ⊕ Area Code ⊕ Phone
 Holmdel ⊕ Holmdel, N. J. 07733 ⊕ 201 ⊕ 949-3000
 Murray Hill ⊕ Murray Hill, N. J. 07974 ⊕ 201 ⊕ 582-6377
 Whippany ⊕ Whippany, N. J. 07981 ⊕ 201 ⊕ 386-3000
 Indian Hill ⊕ Naperville, Illinois 60540 ⊕ 312 ⊕ 690-2000
 .TE

Output:

Bell Labs Locations			
Name	Address	Area Code	Phone
Holmdel	Holmdel, N. J. 07733	201	949-3000
Murray Hill	Murray Hill, N. J. 07974	201	582-6377
Whippany	Whippany, N. J. 07981	201	386-3000
Indian Hill	Naperville, Illinois 60540	312	690-2000

Output:

Some Interesting Places			
Name	Description	Practical Information	
<i>American Museum of Natural History</i>	The collections fill 11.5 acres (Michelin) or 25 acres (MTA) of exhibition halls on four floors. There is a full-sized replica of a blue whale and the world's largest star sapphire (stolen in 1964).	Hours Location Admission Subway Telephone	10-5, ex. Sun 11-5, Wed. to 9 Central Park West & 79th St. Donation: \$1.00 asked AA to 81st St. 212-873-4225
<i>Bronx Zoo</i>	About a mile long and .6 mile wide, this is the largest zoo in America. A lion eats 18 pounds of meat a day while a sea lion eats 15 pounds of fish.	Hours Location Admission Subway Telephone	10-4:30 winter, to 5:00 summer 185th St. & Southern Blvd, the Bronx. \$1.00, but Tu, We, Th free 2, 5 to East Tremont Ave. 212-933-1759
<i>Brooklyn Museum</i>	Five floors of galleries contain American and ancient art. There are American period rooms and architectural ornaments saved from wreckers, such as a classical figure from Pennsylvania Station.	Hours Location Admission Subway Telephone	Wed-Sat, 10-5, Sun 12-5 Eastern Parkway & Washington Ave., Brooklyn. Free 2,3 to Eastern Parkway. 718-638-5000
<i>New-York Historical Society</i>	All the original paintings for Audubon's <i>Birds of America</i> are here, as are exhibits of American decorative arts, New York history, Hudson River school paintings, carriages, and glass paperweights.	Hours Location Admission Subway Telephone	Tues-Fri & Sun, 1-5; Sat 10-5 Central Park West & 77th St. Free AA to 81st St. 212-873-3400

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, 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.



Refer — A Bibliography System

Bill Tuthill

Computing Services
University of California
Berkeley, CA 94720

ABSTRACT

Refer is a bibliography system that supports data entry, indexing, retrieval, sorting, runoff, convenient citations, and footnote or endnote numbering. This document assumes you know how to use some Unix editor, and that you are familiar with the **nroff/troff** text formatters.

The **refer** program is a preprocessor for **nroff/troff**, like **eqn** and **tbl**, except that it is used for literature citations, rather than for equations and tables. Given incomplete but sufficiently precise citations, **refer** finds references in a bibliographic database. The complete references are formatted as footnotes, numbered, and placed either at the bottom of the page, or at the end of a chapter.

A number of ancillary programs make **refer** easier to use. The **addbib** program is for creating and extending the bibliographic database; **sortbib** sorts the bibliography by author and date, or other selected criteria; and **roffbib** runs off the entire database, formatting it not as footnotes, but as a bibliography or annotated bibliography.

Once a full bibliography has been created, access time can be improved by making an index to the references with **indxbib**. Then, the **lookbib** program can be used to quickly retrieve individual citations or groups of citations. Creating this inverted index will speed up **refer**, and **lookbib** will allow you to verify that a citation is sufficiently precise to deliver just one reference.

Introduction

Taken together, the **refer** programs constitute a database system for use with variable-length information. To distinguish various types of bibliographic material, the system uses labels composed of upper case letters, preceded by a percent sign and followed by a space. For example, one document might be given this entry:

```
%A Joel Kies
%T Document Formatting on Unix Using the -ms Macros
%I Computing Services
%C Berkeley
%D 1980
```

Each line is called a field, and lines grouped together are called a record; records are separated from each other by a blank line. Bibliographic information follows the labels, containing data to be used by the **refer** system. The order of fields is not important, except that authors should be entered in the same order as they are listed on the document. Fields can be as long as necessary, and may even be continued on the following line(s).

Data Entry with Addbib

The **addbib** program is for creating and extending bibliographic databases. You must give it the filename of your bibliography:

```
% addbib database
```

Every time you enter **addbib**, it asks if you want instructions. To get them, type **y**; to skip them, type **RETURN**. **Addbib** prompts for various fields, reads from the keyboard, and writes records containing the refer codes to the database. After finishing a field entry, you should end it by typing **RETURN**. If a field is too long to fit on a line, type a backslash (\) at the end of the line, and you will be able to continue on the following line. Note: the backslash works in this capacity only inside **addbib**.

A field will not be written to the database if nothing is entered into it. Typing a minus sign as the first character of any field will cause **addbib** to back up one field at a time. Backing up is the best way to add multiple authors, and it really helps if you forget to add something important. Fields not contained in the prompting skeleton may be entered by typing a backslash as the last character before **RETURN**. The following line will be sent verbatim to the database and **addbib** will resume with the next field. This is identical to the procedure for dealing with long fields, but with new fields, don't forget the % key-letter.

Finally, you will be asked for an abstract (or annotation), which will be preserved as the %X field. Type in as many lines as you need, and end with a control-D (hold down the CTRL button, then press the "d" key). This prompting for an abstract can be suppressed with the **-a** command line option.

After one bibliographic record has been completed, **addbib** will ask if you want to continue. If you do, type **RETURN**; to quit, type **q** or **n** (quit or no). It is also possible to use one of the system editors to correct mistakes made while entering data. After the "Continue?" prompt, type any of the following: **edit**, **ex**, **vi**, or **ed** — you will be placed inside the corresponding editor, and returned to **addbib** afterwards, from where you can either quit or add more data.

If the prompts normally supplied by **addbib** are not enough, are in the wrong order, or are too numerous, you can redefine the skeleton by constructing a promptfile. Create some file, to be named after the **-p** command line option. Place the prompts you want on the left side, followed by a single TAB (control-I), then the refer code that is to appear in the bibliographic database. **Addbib** will send the left side to the screen, and the right side, along with data entered, to the database.

Printing the Bibliography

Sortbib is for sorting the bibliography by author (%A) and date (%D), or by data in other fields. It is quite useful for producing bibliographies and annotated bibliographies, which are seldom entered in strict alphabetical order. It takes as arguments the names of up to 16 bibliography files, and sends the sorted records to standard output (the terminal screen), which may be redirected through a pipe or into a file.

The **-sKEYS** flag to **sortbib** will sort by fields whose key-letters are in the **KEYS** string, rather than merely by author and date. Key-letters in **KEYS** may be followed by a '+' to indicate that all such fields are to be used. The default is to sort by senior author and date (printing the senior author 'last name first'), but **-sA+D** will sort by all authors and then date, and **-sATD** will sort on senior author, then title, and then date.

Roffbib is for running off the (probably sorted) bibliography. It can handle annotated bibliographies — annotations are entered in the %X (abstract) field. **Roffbib** is a shell script that calls **refer -B** and **nroff -mbib**. It uses the macro definitions that reside in **/usr/lib/tmac/tmac.bib**, which you can redefine if you know **nroff** and **troff**. Note that **refer** will print the %H and %O commentaries, but will ignore abstracts in the %X field; **roffbib** will print both fields, unless annotations are suppressed with the **-x** option.



```
.[
  kies document formatting
  %P 10
.]
```

The first line, a partial citation, will find the reference in your bibliography. The second line will insert the page number into the final citation. Ranges of pages may be specified as “%P 56-78”.

When the time comes to run off a paper, you will need to have two files: the bibliographic database, and the paper to format. Use a command line something like one of these:

```
% refer -p database paper | nroff -ms
% refer -p database paper | tbl | nroff -ms
% refer -p database paper | tbl | neqn | nroff -ms
```

If other preprocessors are used, `refer` should precede `tbl`, which must in turn precede `eqn` or `neqn`. The `-p` option specifies a “private” database, which most bibliographies are.

Refer's Command-line Options

Many people like to place references at the end of a chapter, rather than at the bottom of the page. The `-e` option will accumulate references until a macro sequence of the form

```
.[
  $LIST$
.]
```

is encountered (or until the end of file). `Refer` will then write out all references collected up to that point, collapsing identical references. Warning: there is a limit (currently 200) on the number of references that can be accumulated at one time.

It is also possible to sort references that appear at the end of text. The `-sKEYS` flag will sort references by fields whose key-letters are in the `KEYS` string, and permute reference numbers in the text accordingly. It is unnecessary to use `-e` with it, since `-s` implies `-e`. Key-letters in `KEYS` may be followed by a '+' to indicate that all such fields are to be used. The default is to sort by senior author and date, but `-sA+D` will sort on all authors and then date, and `-sA+T` will sort by authors and then title.

`Refer` can also make citations in what is known as the Social or Natural Sciences format. Instead of numbering references, the `-l` (letter ell) flag makes labels from the senior author's last name and the year of publication. For example, a reference to the paper on Inverted Indexes cited above might appear as [Lesk1978a]. It is possible to control the number of characters in the last name, and the number of digits in the date. For instance, the command line argument `-l6,2` might produce a reference such as [Kernig78c].

Some bibliography standards shun both footnote numbers and labels composed of author and date, requiring some keyword to identify the reference. The `-k` flag indicates that, instead of numbering references, key labels specified on the `%L` line should be used to mark references.

The `-n` flag means to not search the default reference file, located in `/usr/dict/papers/Rv7man`. Using this flag may make `refer` marginally faster. The `-an` flag will reverse the first `n` author names, printing Jones, J. A. instead of J. A. Jones. Often `-a1` is enough; this will reverse the names of only the senior author. In some versions of `refer` there is also the `-f` flag to set the footnote number to some predetermined value; for example, `-f23` would start numbering with footnote 23.

Making an Index

Once your database is large and relatively stable, it is a good idea to make an index to it, so that references can be found quickly and efficiently. The `indxbib` program makes an inverted index to the bibliographic database (this program is called `pubindex` in the Bell Labs manual). An inverted index

instead. This is because the %X field is not passed through as a string, but as the body of a paragraph macro.

Another problem arises from authors with foreign names. When a name like “Valéry Giscard d’Estaing” is turned around by the `-a` option of `refer`, it will appear as “d’Estaing, Valéry Giscard,” rather than as “Giscard d’Estaing, Valéry.” To prevent this, enter names as follows:

```
%A Vale\*ry Giscard\0d’Estaing
%A Alexander Csoma\0de\0Ko\*ro\*s
```

(The second is the name of a famous Hungarian linguist.) The backslash-zero is an `nroff/troff` request meaning to insert a digit-width space. It will protect against faulty name reversal, and also against mis-sorting.

Footnote numbers are placed at the end of the line before the `.]` macro. This line should be a line of text, not a macro. As an example, if the line before the `.]` is a `.R` macro, then the `.R` will eat the footnote number. (The `.R` is an `-ms` request meaning change to Roman font.) In cases where the font needs changing, it is necessary to do the following:

```
\flet al.\fR
.[
awk aho kernighan weinberger
.]
```

Now the reference will be to Aho *et al.*² The `\fI` changes to italics, and the `\fR` changes back to Roman font. Both these requests are `nroff/troff` requests, not part of `-ms`. If and when a footnote number is added after this sequence, it will indeed appear in the output.

Internal Details of Refer

You have already read everything you need to know in order to use the `refer` bibliography system. The remaining sections are provided only for extra information, and in case you need to change the way `refer` works.

The output of `refer` is a stream of string definitions, one for each field in a reference. To create string names, percent signs are simply changed to an open bracket, and an `[F` string is added, containing the footnote number. The %X, %Y and %Z fields are ignored; however, the `annobib` program changes the %X to an `.AP` (annotation paragraph) macro. The citation used above yields this intermediate output:

```
.ds [F 1
.]-
.ds [A Mike E. Lesk
.ds [T Some Applications of Inverted Indexes on the Unix System
.ds [J Unix Programmer’s Manual
.ds [I Bell Laboratories
.ds [C Murray Hill, NJ
.ds [D 1978
.ds [V 2a
.nr [T 0
.nr [A 0
.nr [O 0
.][ 1 journal-article
```

These string definitions are sent to `nroff`, which can use the `-ms` macros defined in `/usr/lib/mx/tmac.xref` to take care of formatting things properly. The initializing macro `.]-` precedes

² Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, “Awk — A Pattern Scanning and Processing Language,” *Unix Programmer’s Manual*, vol. 2a, Bell Laboratories, Murray Hill, NJ, 1978.

```
See reference
.{(
partial citation
.}),
```

Note that blanks are significant on these signal lines. If a permanent change in the footnote format is desired, it's best to redefine the [. and .] strings.

Changing the Refer Macros

This section is provided for those who wish to rewrite or modify the **refer** macros. This is necessary in order to make output correspond to specific journal requirements, or departmental standards. First there is an explanation of how new macros can be substituted for the old ones. Then several alterations are given as examples. Finally, there is an annotated copy of the **refer** macros used by **roffbib**.

The **refer** macros for **nroff/troff** supplied by the **-ms** macro package reside in `/usr/lib/mx/tmac.xref`; they are reference macros, for producing footnotes or endnotes. The **refer** macros used by **roffbib**, on the other hand, reside in `/usr/lib/tmac/tmac.bib`; they are for producing a stand-alone bibliography.

To change the macros used by **roffbib**, you will need to get your own version of this shell script into the directory where you are working. These two commands will get you a copy of **roffbib** and the macros it uses: †

```
% cp /usr/lib/tmac/tmac.bib bibmac
```

You can proceed to change **bibmac** as much as you like. Then when you use **roffbib**, you should specify your own version of the macros, which will be substituted for the normal ones

```
% roffbib -m bibmac filename
```

where *filename* is the name of your bibliography file. Make sure there's a space between **-m** and **bibmac**.

If you want to modify the **refer** macros for use with **nroff** and the **-ms** macros, you will need to get a copy of "tmac.xref":

```
% cp /usr/lib/ms/s.ref refmac
```

These macros are much like "bibmac", except they have **.FS** and **.FE** requests, to be used in conjunction with the **-ms** macros, rather than independently defined **.XP** and **.AP** requests. Now you can put this line at the top of the paper to be formatted:

```
.so refmac
```

Your new **refer** macros will override the definitions previously read in by the **-ms** package. This method works only if "refmac" is in the working directory.

Suppose you didn't like the way dates are printed, and wanted them to be parenthesized, with no comma before. There are five identical lines you will have to change. The first line below is the old way, while the second is the new way:

```
.if !"\*(ID"" , \*(ID\c
.if !"\*(ID"" \& (\*(ID)\c
```

In the first line, there is a comma and a space, but no parentheses. The "\c" at the end of each line indicates to **nroff** that it should continue, leaving no extra space in the output. The "&" in the second line is the do-nothing character; when followed by a space, a space is sent to the output.

If you need to format a reference in the style favored by the Modern Language Association or Chicago University Press, in the form (city: publisher, date), then you will have to change the middle

Some Applications of Inverted Indexes on the UNIX System

M. E. Lesk

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction.

The UNIX† system has many utilities (e.g. *grep*, *awk*, *lex*, *egrep*, *fgrep*, ...) to search through files of text, but most of them are based on a linear scan through the entire file, using some deterministic automaton. This memorandum discusses a program which uses inverted indexes¹ and can thus be used on much larger data bases.

As with any indexing system, of course, there are some disadvantages; once an index is made, the files that have been indexed can not be changed without remaking the index. Thus applications are restricted to those making many searches of relatively stable data. Furthermore, these programs depend on hashing, and can only search for exact matches of whole keywords. It is not possible to look for arithmetic or logical expressions (e.g. "date greater than 1970") or for regular expression searching such as that in *lex*.²

Currently there are two uses of this software, the *refer* preprocessor to format references, and the *lookall* command to search through all text files on the UNIX system.‡

The remaining sections of this memorandum discuss the searching programs and their uses. Section 2 explains the operation of the searching algorithm and describes the data collected for use with the *lookall* command. The more important application, *refer* has a user's description in section 3. Section 4 goes into more detail on reference files for the benefit of those who wish to add references to data bases or write new *troff* macros for use with *refer*. The options to make *refer* collect identical citations, or otherwise relocate and adjust references, are described in section 5.

2. Searching.

The indexing and searching process is divided into two phases, each made of two parts. These are shown below.

A. Construct the index.

- (1) Find keys — turn the input files into a sequence of tags and keys, where each tag identifies a distinct item in the input and the keys for each such item are the strings under which it is to be indexed.
- (2) Hash and sort — prepare a set of inverted indexes from which, given a set of keys, the appropriate item tags can be found quickly.

B. Retrieve an item in response to a query.

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

¹ D. Knuth, *The Art of Computer Programming: Vol. 3, Sorting and Searching*, Addison-Wesley, Reading, Mass., 1977. See section 6.5.

² M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, October 1975. Reprinted as PS1:16 in *UNIX Programmer's Manual*, Usenix Association, (1986).

‡ *lookall* is not part of the Berkeley UNIX distribution.

false drops, since items referenced by the correct hash codes need not actually have contained the correct keys. Normally, if there are several keys in the query, there are not likely to be many false drops in the final combined list even though each hash code is somewhat ambiguous. The actual tags are then obtained from the tag file, and to guard against the possibility that an item has false-dropped on some hash code in the query, the original items are normally obtained from the delivery program (4) and the query keys checked against them by string comparison.

Usually, therefore, the check for bad drops is made against the original file. However, if the key derivation procedure is complex, it may be preferable to check against the keys fed to program (2). In this case the optional key file which contains the keys associated with each item is generated, and the item tag is supplemented by a string

```
;start,length
```

which indicates the starting byte number in the key file and the length of the string of keys for each item. This file is not usually necessary with the present key-selection program, since the keys always appear in the original document.

There is also an option (*-Cn*) for coordination level searching. This retrieves items which match all but *n* of the query keys. The items are retrieved in the order of the number of keys that they match. Of course, *n* must be less than the number of query keys (nothing is retrieved unless it matches at least one key).

As an example, consider one set of 4377 references, comprising 660,000 bytes. This included 51,000 keys, of which 5,900 were distinct keys. The hash table is kept full to save space (at the expense of time); 995 of 997 possible hash codes were used. The total set of index files (no key file) included 171,000 bytes, about 26% of the original file size. It took 8 minutes of processor time to hash, sort, and write the index. To search for a single query with the resulting index took 1.9 seconds of processor time, while to find the same paper with a sequential linear search using *grep* (reading all of the tags and keys) took 12.3 seconds of processor time.

We have also used this software to index all of the English stored on our UNIX system. This is the index searched by the *lookall* command. On a typical day there were 29,000 files in our user file system, containing about 152,000,000 bytes. Of these 5,300 files, containing 32,000,000 bytes (about 21%) were English text. The total number of 'words' (determined mechanically) was 5,100,000. Of these 227,000 were selected as keys; 19,000 were distinct, hashing to 4,900 (of 5,000 possible) different hash codes. The resulting inverted file indexes used 845,000 bytes, or about 2.6% of the size of the original files. The particularly small indexes are caused by the fact that keys are taken from only the first 50 non-common words of some very long input files.

Even this large *lookall* index can be searched quickly. For example, to find this document by looking for the keys "lesk inverted indexes" required 1.7 seconds of processor time and system time. By comparison, just to search the 800,000 byte dictionary (smaller than even the inverted indexes, let alone the 27,000,000 bytes of text files) with *grep* takes 29 seconds of processor time. The *lookall* program is thus useful when looking for a document which you believe is stored on-line, but do not know where. For example, many memos from our center are in the file system, but it is often difficult to guess where a particular memo might be (it might have several authors, each with many directories, and have been worked on by a secretary with yet more directories). Instructions for the use of the *lookall* command are given in the manual section, shown in the appendix to this memorandum.

The only indexes maintained routinely are those of publication lists and all English files. To make other indexes, the programs for making keys, sorting them, searching the indexes, and delivering answers must be used. Since they are usually invoked as parts of higher-level commands, they are not in the default command directory, but are available to any user in the directory */usr/lib/refer*. Three programs are of interest: *mkey*, which isolates keys from input files; *inv*, which makes an index from a set of keys; and *hunt*, which searches the index and delivers the items. Note that the two parts of the retrieval phase are combined into one program, to avoid the excessive system work and delay which would result from running these as separate processes.

- p Pipe into the sort program, rather than writing a temporary input file. This saves disk space and spends processor time.
- v Verbose mode; print a summary of the number of keys which finished indexing.

About half the time used in *inv* is in the contained sort. Assuming the sort is roughly linear, however, a guess at the total timing for *inv* is 250 keys per second. The space used is usually of more importance: the entry file uses four bytes per possible hash (note the *-h* option), and the tag file around 15-20 bytes per item indexed. Roughly, the posting file contains one item for each key instance and one item for each possible hash code; the items are two bytes long if the tag file is less than 65536 bytes long, and the items are four bytes wide if the tag file is greater than 65536 bytes long. Note that to minimize storage, the hash tables should be over-full; for most of the files indexed in this way, there is no other real choice, since the *entry* file must fit in memory.

Searching and Retrieving. The *hunt* program retrieves items from an index. It combines, as mentioned above, the two parts of phase (B): search and delivery. The reason why it is efficient to combine delivery and search is partly to avoid starting unnecessary processes, and partly because the delivery operation must be a part of the search operation in any case. Because of the hashing, the search part takes place in two stages: first items are retrieved which have the right hash codes associated with them, and then the actual items are inspected to determine false drops, i.e. to determine if anything with the right hash codes doesn't really have the right keys. Since the original item is retrieved to check on false drops, it is efficient to present it immediately, rather than only giving the tag as output and later retrieving the item again. If there were a separate key file, this argument would not apply, but separate key files are not common.

Input to *hunt* is taken from the standard input, one query per line. Each query should be in *mkey -s* output format; all lower case, no punctuation. The *hunt* program takes one argument which specifies the base name of the index files to be searched. Only one set of index files can be searched at a time, although many text files may be indexed as a group, of course. If one of the text files has been changed since the index, that file is searched with *fgrep*; this may occasionally slow down the searching, and care should be taken to avoid having many out of date files. The following option arguments are recognized by *hunt*:

- a Give all output; ignore checking for false drops.
- Cn Coordination level *n*; retrieve items with not more than *n* terms of the input missing; default *C0*, implying that each search term must be in the output items.
- F[ynd] “-Fy” gives the text of all the items found; “-Fn” suppresses them. “-Fd” where *d* is an integer gives the text of the first *d* items. The default is *-Fy*.
- g Do not use *fgrep* to search files changed since the index was made; print an error comment instead.
- i *string* Take *string* as input, instead of reading the standard input.
- l *n* The maximum length of internal lists of candidate items is *n*; default 1000.
- o *string* Put text output (“-Fy”) in *string*; of use *only* when invoked from another program.
- p Print hash code frequencies; mostly for use in optimizing hash table sizes.
- T[ynd] “-Ty” gives the tags of the items found; “-Tn” suppresses them. “-Td” where *d* is an integer gives the first *d* tags. The default is *-Tn*.
- t *string* Put tag output (“-Ty”) in *string*; of use *only* when invoked from another program.

The timing of *hunt* is complex. Normally the hash table is overfull, so that there will be many false drops on any single term; but a multi-term query will have few false drops on all terms. Thus if



citations searched may be tailored to each system, and individual users may specify their own citation files. On our system, the default data base is accumulated from the publication lists of the members of our organization, plus about half a dozen personal bibliographies that were collected. The present total is about 4300 citations, but this increases steadily. Even now, the data base covers a large fraction of local citations.

For example, the reference for the *eqn* paper above was specified as

```
...
preprocessor like
.I eqn.
.[
kernighan cherry acm 1975
.]
It scans its input looking for items
...
```

This paper was itself printed using *refer*. The above input text was processed by *refer* as well as *tbl* and *troff* by the command

```
refer memo-file | tbl | troff -ms
```

and the reference was automatically translated into a correct citation to the ACM paper on mathematical typesetting.

The procedure to use to place a reference in a paper using *refer* is as follows. First, use the *lookbib* command to check that the paper is in the data base and to find out what keys are necessary to retrieve it. This is done by typing *lookbib* and then typing some potential queries until a suitable query is found. For example, had one started to find the *eqn* paper shown above by presenting the query

```
$ lookbib
kernighan cherry
(EOT)
```

lookbib would have found several items; experimentation would quickly have shown that the query given above is adequate. Overspecifying the query is of course harmless. A particularly careful reader may have noticed that "acm" does not appear in the printed citation; we have supplemented some of the data base items with common extra keywords, such as common abbreviations for journals or other sources, to aid in searching.

If the reference is in the data base, the query that retrieved it can be inserted in the text, between *.]* and *.]* brackets. If it is not in the data base, it can be typed into a private file of references, using the format discussed in the next section, and then the *-p* option used to search this private file. Such a command might read (if the private references are called *myfile*)

```
refer -p myfile document | tbl | eqn | troff -ms . . .
```

where *tbl* and/or *eqn* could be omitted if not needed. The use of the *-ms* macros⁴ or some other macro package, however, is essential. *Refer* only generates the data for the references; exact formatting is done by some macro package, and if none is supplied the references will not be printed.

By default, the references are numbered sequentially, and the *-ms* macros format references as footnotes at the bottom of the page. This memorandum is an example of that style. Other possibilities are discussed in section 5 below.

⁴ M. E. Lesk, *Typing Documents on UNIX and GCOS: The -ms Macros for Troff*, 1977. Revised version reprinted as USD:20 in *UNIX User's Manual*, Usenix Association, (1986).

```

%T Bounds on the Complexity of the Maximal
Common Subsequence Problem
%Z ctr127
%A A. V. Aho
%A D. S. Hirschberg
%A J. D. Ullman
%J J. ACM
%V 23
%N 1
%P 1-12
%M abcd-78
%D Jan. 1976

```

Order is irrelevant, except that authors are shown in the order given. The output of *refer* is a stream of string definitions, one for each of the fields of each reference, as shown below.

```

.]-
.ds [A authors' names ...
.ds [T title ...
.ds [J journal ...
...
.][ type-number

```

The special macro `.]-` precedes the string definitions and the special macro `.]` follows. These are changed from the input `.[` and `.]` so that running the same file through *refer* again is harmless. The `.]-` macro can be used by the macro package to initialize. The `.]` macro, which should be used to print the reference, is given an argument *type-number* to indicate the kind of reference, as follows:

Value	Kind of reference
1	Journal article
2	Book
3	Article within book
4	Technical report
5	Bell Labs technical memorandum
0	Other

The reference is flagged in the text with the sequence

```
\*([.number\*(.])
```

where *number* is the footnote number. The strings `[.` and `.]` should be used by the macro package to format the reference flag in the text. These strings can be replaced for a particular footnote, as described in section 5. The footnote number (or other signal) is available to the reference macro `.]` as the string register `[F`.

In some cases users wish to suspend the searching, and merely use the reference macro formatting. That is, the user doesn't want to provide a search key between `.[` and `.]` brackets, but merely the reference lines for the appropriate document. Alternatively, the user can wish to add a few fields to those in the reference as in the standard file, or override some fields. Altering or replacing fields, or supplying whole references, is easily done by inserting lines beginning with `%`; any such line is taken as direct input to the reference processor rather than keys to be searched. Thus

```

.[
key1 key2 key3 ...
%Q New format item
%R Override report name
.]

```

makes the indicated changes to the result of searching for the keys. All of the search keys must be

is encountered. Thus, to place references at the end of a paper, the user would run *refer* with the *-e* option and place the above \$LIST\$ commands after the last line of the text. *Refer* will then move all the references to that point. To aid in formatting the collected references, *refer* writes the references preceded by the line

```
.|<
```

and followed by the line

```
.|>
```

to invoke special macros before and after the references.

Another possible option to *refer* is the *-s* option to specify sorting of references. The default, of course, is to list references in the order presented. The *-s* option implies the *-e* option, and thus requires a

```
.|
$LIST$
.|
```

entry to call out the reference list. The *-s* option may be followed by a string of letters, numbers, and '+' signs indicating how the references are to be sorted. The sort is done using the fields whose key-letters are in the string as sorting keys; the numbers indicate how many of the fields are to be considered, with '+' taken as a large number. Thus the default is *-sAD* meaning "Sort on senior author, then date." To sort on all authors and then title, specify *-sA+T*. And to sort on two authors and then the journal, write *-sA2J*.

Other options to *refer* change the signal or label inserted in the text for each reference. Normally these are just sequential numbers, and their exact placement (within brackets, as superscripts, etc.) is determined by the macro package. The *-l* option replaces reference numbers by strings composed of the senior author's last name, the date, and a disambiguating letter. If a number follows the *l* as in *-l3* only that many letters of the last name are used in the label string. To abbreviate the date as well the form *-lm,n* shortens the last name to the first *m* letters and the date to the last *n* digits. For example, the option *-l3,2* would refer to the *eqn* paper (reference 3) by the signal *Ker75a*, since it is the first cited reference by Kernighan in 1975.

A user wishing to specify particular labels for a private bibliography may use the *-k* option. Specifying *-kx* causes the field *x* to be used as a label. The default is *L*. If this field ends in *-*, that character is replaced by a sequence letter; otherwise the field is used exactly as given.

If none of the *refer*-produced signals are desired, the *-b* option entirely suppresses automatic text signals.

If the user wishes to override the *-ms* treatment of the reference signal (which is normally to enclose the number in brackets in *nroff* and make it a superscript in *troff*) this can be done easily. If the lines *.|* or *.|* contain anything following these characters, the remainders of these lines are used to surround the reference signal, instead of the default. Thus, for example, to say "See reference (2)." and avoid "See reference.²" the input might appear

```
See reference
.| (
imprecise citation ...
.|).
```

Note that blanks are significant in this construction. If a permanent change is desired in the style of reference signals, however, it is probably easier to redefine the strings *.|* and *.|* (which are used to bracket each signal) than to change each citation.

Although normally *refer* limits itself to retrieving the data for the reference, and leaves to a macro package the job of arranging that data as required by the local format, there are two special options for rearrangements that can not be done by macro packages. The *-c* option puts fields into all upper case (CAPS-SMALL CAPS in *troff* output). The key-letters indicated what information is to be

BIB – A Program for Formatting Bibliographies

Timothy A. Budd

The University of Arizona
Department of Computer Science
Tucson, Arizona 85721

Bib is a program for collecting and formatting reference lists in documents. It is a preprocessor to the *nroff*/*troff* typesetting systems, (much like the *tbl* [4] and *eqn* [2] programs) and an alternative to the *refer* [3] bibliography program. *Bib* takes two inputs: a document to be formatted and a library of references. Imprecise citations in the source document are replaced by more conventional citation strings, the appropriate references are selected from the reference file, and commands are generated to format both citation and the referenced item in the bibliography.

An imprecise citation is a list of words surrounded by the characters [.]. Words (which are truncated to six letters) in the imprecise citation are matched against entries in the reference file, and if an entry is found that matches all words, that reference is used. For example:

In Brooks's interesting book [. brooks mythical.] various reasons ...

Multiple citations are indicated by simply placing a comma in the imprecise citation:

In [.kernig tools, kernig elements.], Kernighan and Plauger have ...

Embedded newlines, tabs and extra blanks within the imprecise citation are ignored.

Judicious use of the K (keyword) field in references in the database can simplify citations considerably. Also additional information can be placed into citations by surrounding text with curly braces. The additional information is inserted verbatim into the citation, e.g. [1, Chapter 6]. Note that it may be desirable to use non-breakable spaces, in order that the citation not be split across a line boundary by *troff*, for example:

For a description of LR parsing, see [.dragon {, \ Chapter 6}.] by Aho and Ullman.

An alternative citation style can be used by surrounding the imprecise citation with { . and .}. Most document styles just give the raw citation, without the braces, in this case. This is useful, for example, to refer to citations in running text.

For a discussion of this point, see reference { .dragon.}.

The algorithm used by *bib* scans the source input in two passes. In the first pass, references are collected and the location of citations marked. In the second pass, these marks are replaced by the appropriate citation, and the entire list of references is dumped following a call on the macro [.] . This macro is left untouched. However, this can be altered to achieve other typographic effects.

An exception to this process is made in those instances where references are indicated in footnotes. In this case the macro that generates the reference is placed immediately after each line in which the reference is cited.

the citation (i.e., Knu79 becomes Knu79a and Knu79b). As noted previously, this can be altered by using the F field.

For the purposes of sorting by author, the last name is taken to be the last word of the name field. This means some care must be taken when names contain embedded blanks, such as in 'Hartley Rogers, Jr.' or 'Mary-Claire van Leunen'. In these cases a concealed space (\) should be used, as in 'Hartley Rogers,\ Jr.'.

bib knows very little about *troff* usage or syntax. This can sometimes be useful. For example, to cause an entry to appear in a reference list without having it explicitly cited in the text the citation can be placed in a *troff* comment.

.\ " [.imprecise citation.]

It is also possible to embed *troff* commands within a reference definition. See 'abbreviations' in the section 'Reference Format Designer's Guide' for an example.

In some styles (superscripts) periods and commas should precede the citation while spaces follow. In other styles (brackets) these rules are reversed. If a period, comma or space immediately precedes a citation, it will be moved to the appropriate location for the particular reference style being used. This movement is not done for citations given in the alternative style.

The following is a complete list of options for *bib*:

- aa reduce author's first names to abbreviations.
- arnum reverse the first *num* author's names. If *num* is omitted all names are reversed.
- ax print authors last names in Caps-Small Caps style. For example Budd becomes BUDD.
- cstr build citations according to the template *str*. See the reference format designer's guide for more information on templates.
- ea abbreviate editors' names
- ex places editors' names in Caps-Small Caps style. (see -x)
- ernum reverse the first *num* editors' names. If *num* is omitted all editors' names are reversed.
- f instead of dumping references following the call on .[,], dump each reference immediately following the line on which the citation is placed (used for footnoted references).
- h hyphenate runs of three or more contiguous references in the citation string. (eg 2,3,4,5 becomes 2-5). This is most useful for numeric citation styles, but works generally. The -h option implies the -o option.
- i file
- ifile include and process the indicated file. This is useful for including a private file of string definitions.
- nstr turn off the indicated options. *str* must be composed of the characters *ashorx*.
- o sort contiguous citations according to the order given by the reference list. (This option defaults on).
- p file
- pfile instead of searching the file INDEX, search the indicated reference file(s) before searching the system file. Multiple files are separated by commas.
- sstr sort references according to the template *str*.
- t type
- ttype use the standard macros and switch settings to generate citations and references in the indicated style.

Sometimes a book (particularly old books) will have no listed publisher. The reference entry must still have an I field.

```
%A R. Colt Hoare
%T A Tour through the Island of Elba
%I (no listed publisher)
%C London
%D 1814
```

If a reference database contains entries from many people (such as a departmental-wide database), the W field can be used to indicate where the referenced item can be found; using the initials of the owner, for example. Any entry style can take a W field, since this field is not used in formatting the reference.

The K field is used to define general subject categories for an entry. This is useful in locating all entries pertaining to a specific subject area. Note the use of the backslash, to indicate the last name is Van Tassel, and not simply Tassel.

```
%A Dennie Van\ Tassel
%T Program Style, Design, Efficiency,
Debugging and Testing
%I PRHALL
%D 1978
%W tab
%K testing debugging
```

Journal article

The only requirement for a journal article is that it have a journal name and a volume number. Usually journal articles also have authors, titles, page numbers, and a date of publication. They may also have numbers, and, less frequently, a publisher. (Generally, publishers are only listed for obscure journals).

Note that string names (such as CACM for *Communications of the ACM*) are defined for most major journals. There are also string names for the months of the year, so that months can be abbreviated to the first three letters. Note also in this example the use of the K field to define a short name (hru) that can be used in searching for the reference.

```
%A M. A. Harrison
%A W. L. Ruzzo
%A J. D. Ullman
%T Protection in Operating Systems
%J CACM
%V 19
%N 8
%P 461-471
%D AUG 1976
%K hru
```

Article in conference proceedings

An article from a conference is printed as though it were a journal article and the journal name was the name of the conference. Note that string names (SOSP) are also defined for the major conferences (Symposium on Operating System Principles).

```
%A M. Bishop
%A L. Snyder
%T The Transfer of Information and Authority
```



%O (Discussed in Glib [32])

Compilations

A compilation is the work of several authors gathered together by an editor into a book. The reference format is the same as for a book, with the editor(s) taking the place of the author.

%E R. A. DeMillo
 %E D. P. Dobkin
 %E A. K. Jones
 %E R. J. Lipton
 %T Foundations of Secure Computation
 %I ACPRESS
 %D 1978

Technical Reports

A technical report must have a report number. They usually have authors, titles, dates and an issuing institution (the I field is used for this). They may also have a city and a government issue number. Again string values (UATR for 'University of Arizona Technical Report') will frequently simplify typing references.

%A T. A. Budd
 %T An APL Compiler
 %R UATR 81-17
 %C Tucson, Arizona
 %D 1981

If the institution name is not part of the technical report number, then the institution should be given separately.

%A Douglas Baldwin
 %A Frederick Sayward
 %T Heuristics for Determining Equivalence of Program Mutations
 %R Technical Report Number 161
 %I Yale University
 %D 1979

PhD Thesis

A PhD thesis is listed as if it were a book, and the institution granting the degree the publisher.

%A Martin Brooks
 %T Automatic Generation of Test Data for
 Recursive Programs Having Simple Errors
 %I PhD Thesis, Stanford University
 %D 1980

Some authors prefer to treat Master's and Bachelor theses similarly, although most references on style instruct say to treat a Master's degree as an article or as a report.

%A A. Snyder
 %T A Portable Compiler for the Language C
 %R Master's Thesis
 %I M.I.T.
 %D 1974

F

The F command indicates that references are to be dumped immediately after a line containing a citation, such as when the references are to be placed in footnotes.

S *template*

The S command indicates references are to be sorted before being dumped. The comparison used in sorting is based on the *template*. See the discussion on sorting (below) for an explanation of templates.

C *template*

The *template* is used as a model in constructing citations. See the discussion below.

D *word definition*

The word-definition pair is placed into a table. Before each reference is dumped it is examined for the occurrence of these words. Any occurrence of a word from this table is replaced by the definition, which is then rescanned for other words. Words are limited to alphanumeric characters, ampersand and underscore.

Definitions can extend over multiple lines by ending lines with a backslash (\). The backslash will be removed, and the definition, including the newline and the next line, will be entered into the table. This is useful for including several fields as part of a single definition (city names can be included as part of a definition for a publishing house, for example).

I *filename*

The indicated file is included at the current point. The included file may contain other formatting commands.

H

Three or more contiguous citations that refer to adjacent items in the reference list are replaced by a hyphenated string. For example, the citation 2,3,4,5 would be replaced by 2-5. This is most useful with numeric citations. The H option implies the O option.

O

Contiguous citations are sorted according to the order given by the reference list.

R *number*

The first *number* author's names are reversed on output (i.e. T. A. Budd becomes Budd, T. A.). If number is omitted all names are reversed.

T *str*

The *str* is a list of field names. Each time a definition string for a named field is produced, a second string containing just the last character will also be generated. See 'Trailing characters', below.

X

Authors' last names are to be printed in Caps/Small Caps format (i.e., Budd becomes BUDD).

The first line in the format file that does not match a format command causes that line, and all subsequent lines, to be immediately copied to the output.

is used to specify a format consisting of the authors' last names, or the senior author followed by the text 'et al' if more than four authors are listed. The fields '4' through '9' are reserved to be used to specify formats that cannot be produced using templates. These will be implemented either as local modifications to *bib* or in future releases.

In order to postpone the inevitable clash of local changes versus new releases, it is suggested that local formatting styles use numbers starting at 9 and working downward.

Each object can be followed by either of the letters 'u' or 'l' and the field will be printed in all upper or all lower case, respectively.

If necessary for disambiguating, the character '@' can be used as a separator between objects in the citation template. Any text which should be inserted into the citation uninterpreted should be surrounded by either {} or <> pairs.

Citation Formatting

In the output, each citation is surrounded by the strings *{[and *{]} (*{[and *{]} in the alternative style). Multiple citations are separated by the string *(,). The text portion of a format file should contain *troff* definitions for these strings to achieve the appropriate typographic effect.

Citations that are preceded by a period, comma, space or other punctuation are surrounded by string values for formatting the punctuation in the appropriate location. Again, *troff* commands should be given to insure the appropriate values are produced.

The following table summarizes the string values that must be defined to handle citations.

[[]]	Standard citation beginning and ending
{[]}	Alternate citation beginning and ending
[.]	Period before and after citation
[,]	Comma before and after citation
[? ?]	Question mark before and after citation
[! !]	Exclamation Point before and after citation
[: :]	Colon before and after citation
[; ;]	Semi-Colon before and after citation
[" "]	Double Quote before and after citation
[' ']	Single Quote before and after citation
[< >]	Space before and after citation
],	Multiple citation separator
]-	Separator for a range of citations

Name Formatting

Authors' (and editors') names can be abbreviated, reversed, and/or printed in Caps-small Caps format. In producing the string values for an author, formatting strings are inserted to give the macro writer greater flexibility in producing the final output. Currently the following strings are used:

- a] gap between successive initials
- b] comma between last name and initial in reversed text
- c] comma between authors
- n] *and* between two authors
- m] *and* between last two authors
- p] period following initial

For example, suppose the name 'William E. Howden' is abbreviated and reversed. It will come out looking like

Howden\[b]W\[p]\[a]E\[p]

should test this value before generating punctuation.

Abbreviations

The algorithm used to generate abbreviations from first names is fairly simple: Each word in the first name field that begins with a capital is reduced to that capital letter followed by a period. In some cases, this may not be sufficient. For example, suppose Ole-Johan Dahl should be abbreviated 'O-J. Dahl'. The only way to achieve this (short of editing the output) is to include *troff* commands in the reference file that alter the strings produced by *bib*, as in the following

```
...
%A Ole-Johan Dahl
.ds [A O-J. Dahl
...
```

In fact, any *troff* commands can be entered in the middle of a reference entry, and the commands are copied uninterpreted to the output. For example, the user may wish to have a switch indicating whether the name is to be abbreviated or not:

```
...
%A Ole-Johan Dahl
.if \n(i[ .ds [A O-J. Dahl
...
```

An Example

Figure 1 shows the format file for the standard alphabetic format. The *sort* command indicates that sorting is to be done by senior author, followed by the last two digits of the date. The citation template indicates that citations will be the three character sequence described in the section of citations followed by the last two characters of the date (i.e. AHU79, for example).

```
# standard alphabetic format
SAD-2
C2D-2
I BMACLIB/bibinc.fullnames
I BMACLIB/bibinc.std
```

Figure 1

The two *I* commands include two files. The first is a file of definitions for common strings, such as dates and journal names. A portion of this file is shown in figure 2. Note that a no-op has been inserted into the definition string for *BIT* in order to avoid further expansion when the definition is rescanned.

The second file is a sequence of *troff* macros for formatting the references. The beginning of this file is shown in figure 3.

On the basis of some simple rules (the presence or absence of certain fields) the document is identified as one of five different types, and a call made on a different macro for each type. This is shown in figure 4.

Finally figure 5 shows the macro for one of those different types, in this case the book formatting macro.

```

...
.de 2[ \ " book
.if !\*(F"" .p[ \*(F
.if !\*(A"" \*(A,
.if !\*(T"" \f2\*(T,\f1
\*(I\c
.if !\*(C"" , \*(C\c
.if !\*(D"" \& (\*(D)\c
\&.
.if !\*(G"" Gov't. ordering no. \*(G.
.if !\*(O"" \*(O
.-
..

```

Figure 5

Acknowledgements

bib was inspired by *refer*, written by M. Lesk.

1. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1977.
2. B. W. Kernighan and L. L. Cherry, A System for Typesetting Mathematics, *Comm. of the ACM* 18, 3 (Mar. 1978), 151-156.
3. M. E. Lesk, Some Applications of Inverted Indexes on the UNIX System, Bell Laboratories Computing Science Technical Report 69, June 1978.
4. M. E. Lesk, Tbl - A Program to Format Tables, *Unix Programmer's Manual, Vol 2A, .*

OPL7 Conference Record of the Seventh POPL
 OPL8 Conference Record of the Eighth POPL
 OPL9 Conference Record of the Ninth POPL
 OPL10 Conference Record of the Tenth POPL
 OPL11 Conference Record of the Eleventh POPL
 ROC Proceedings
 OSP SYMP on Operating System Principles
 TOC Annual ACM SYMP on Theory of Computing
 YMP Symposium
 √JCC PROC Western Joint Computer CONF

onger place names

TLHO Bell Laboratories
 TLMH Bell Laboratories
 MU Carnegie-Mellon University
 MUCS Computer Science Department, Carnegie-Mellon University
 DG Data General
 MITAI MIT Artificial Intelligence Laboratory
 MITLCS MIT Laboratory for Computer Science
 UCS Computer Science Department, Stanford University
 UCSL Computer Systems Lab., Stanford Electronics Lab., Dept. of Electrical Engineering and Computer Science
 UEE Department of Electrical Engineering, Stanford University
 UM Technische Universität München
 JCB University of California, Berkeley
 JCBCS Computer Science Division, EECS, UCB
 JCBERLERL, EECS, UCB

hort place names

ORP Corporation
 SD Computer Science Department
 CS Department of Computer Science
 DEPT Department
 DISS Dissertation
 R Technical Report
 UTR University of Arizona Technical Report
 UNIV University
 RL Electronics Research Laboratory

CSPRESS Computer Science Press
 DIGITAL Digital Press
 ELSEVIER American Elsevier
 FREEMAN W. H. Freeman and Company
 GPO U. S. Government Printing Office
 HOLT Holt, Rinehart, and Winston
 IEEEP IEEE Press
 MCGRAW McGraw-Hill
 MGHILL McGraw-Hill
 MITP MIT Press
 NHOLL North-Holland
 NYC New York, NY
 PRENTICE Prentice Hall
 PRHALL Prentice Hall
 SPRINGER Springer Verlag
 SRA Science Research Associates
 WILEY John Wiley & Sons
 WINTH Winthrop Publishers

onths of the year

AN January
 EB February
 FAR March
 PR April
 IAY May
 JN June
 UL July
 UG August
 EP September
 CT October
 .OV November
 DEC December

ublishers

CADEMIC Academic Press
 CPRESS Academic Press
 DDISON Addison Wesley
 NSI American National Standards Institute



Writing Tools - The STYLE and DICTION Programs

L. L. Cherry

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

W. Vesterman

Livingston College
Rutgers University

ABSTRACT

Text processing systems are now in heavy use in many companies to format documents. With many documents stored on line, it has become possible to use computers to study writing style itself and to help writers produce better written and more readable prose. The system of programs described here is an initial step toward such help. It includes programs and a data base designed to produce a stylistic profile of writing at the word and sentence level. The system measures readability, sentence and word length, sentence type, word usage, and sentence openers. It also locates common examples of wordy phrasing and bad diction. The system is useful for evaluating a document's style, locating sentences that may be difficult to read or excessively wordy, and determining a particular writer's style over several documents.

1. Introduction

Computers have become important in the document preparation process, with programs to check for spelling errors and to format documents. As the amount of text stored on line increases, it becomes feasible and attractive to study writing style and to attempt to help the writer in producing readable documents. The system of writing tools described here is a first step toward such help. The system includes programs and a data base to analyze writing style at the word and sentence level. We use the term "style" in this paper to describe the results of a writer's particular choices among individual words and sentence forms. Although many judgements of style are subjective, particularly those of word choice, there are some objective measures that experts agree lead to good style. Three programs have been written to measure some of the objectively definable characteristics of writing style and to identify some commonly misused or unnecessary phrases. Although a document that conforms to the stylistic rules is not guaranteed to be coherent and readable, one that violates all of the rules is likely to be difficult or tedious to read. The program STYLE calculates readability, sentence length variability, sentence type, word usage and sentence openers at a rate of about 400 words per second on a PDP11/70 running the UNIX† Operating System. It assumes that the sentences are well-formed, i. e. that each sentence has a verb and that the subject and verb agree in number. DICTION identifies phrases that are either bad usage or unnecessarily wordy. EXPLAIN acts as a thesaurus for the phrases found by DICTION. Sections 2, 3, and 4 describe the programs; Section 5 gives the results on a cross-section of technical documents; Section 6 discusses accuracy and problems; Section 7 gives implementation details.

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

2.1. What is a sentence?

Readers of documents have little trouble deciding where the sentences end. People don't even have to stop and think about uses of the character "." in constructions like 1.25, A. J. Jones, Ph.D., i. e., or etc. . When a computer reads a document, finding the end of sentences is not as easy. First we must throw away the printer's marks and formatting commands that litter the text in computer form. Then STYLE defines a sentence as a string of words ending in one of:

. ! ? / .

The end marker "." may be used to indicate an imperative sentence. Imperative sentences that are not so marked are not identified as imperative. STYLE properly handles numbers with embedded decimal points and commas, strings of letters and numbers with embedded decimal points used for naming computer file names, and the common abbreviations listed in Appendix 1. Numbers that end sentences, like the preceding sentence, cause a sentence break if the next word begins with a capital letter. Initials only cause a sentence break if the next word begins with a capital and is found in the dictionary of function words used by PARTS. So the string

J. D. JONES

does not cause a break, but the string

... system H. The ...

does. With these rules most sentences are broken at the proper place, although occasionally either two sentences are called one or a fragment is called a sentence. More on this later.

2.2. Readability Grades

The first section of STYLE output consists of four readability indices. As Klare points out in [3] readability indices may be used to estimate the reading skills needed by the reader to understand a document. The readability indices reported by STYLE are based on measures of sentence and word lengths. Although the indices may not measure whether the document is coherent and well organized, experience has shown that high indices seem to be indicators of stylistic difficulty. Documents with short sentences and short words have low scores; those with long sentences and many polysyllabic words have high scores. The 4 formulae reported are Kincaid Formula [4], Automated Readability Index [5], Coleman-Liau Formula [6] and a normalized version of Flesch Reading Ease Score [7]. The formulae differ because they were experimentally derived using different texts and subject groups. We will discuss each of the formulae briefly; for a more detailed discussion the reader should see [3].

The Kincaid Formula, given by:

$$Reading_Grade = 11.8 * syl_per_wd + .39 * wds_per_sent - 15.59$$

was based on Navy training manuals that ranged in difficulty from 5.5 to 16.3 in reading grade level. The score reported by this formula tends to be in the mid-range of the 4 scores. Because it is based on adult training manuals rather than school book text, this formula is probably the best one to apply to technical documents.

The Automated Readability Index (ARI), based on text from grades 0 to 7, was derived to be easy to automate. The formula is:

$$Reading_Grade = 4.71 * let_per_wd + .5 * wds_per_sent - 21.43$$

ARI tends to produce scores that are higher than Kincaid and Coleman-Liau but are usually slightly lower than Flesch.

The Coleman-Liau Formula, based on text ranging in difficulty from .4 to 16.3, is:

$$Reading_Grade = 5.89 * let_per_wd - .3 * sent_per_100_wds - 15.8$$

Of the four formulae this one usually gives the lowest grade when applied to technical documents.

2. A complex sentence has one independent clause and one dependent clause, each with one verb. Complex sentences are found by identifying sentences that contain either a subordinate conjunction or a clause beginning with words like "that" or "who". The preceding sentence has such a clause.
3. A compound sentence has more than one verb and no dependent clause. Sentences joined by "and" are also counted as compound.
4. A compound-complex sentence has either several dependent clauses or one dependent clause and a compound verb in either the dependent or independent clause.

Even using these broader definitions, simple sentences dominate many of the technical documents that have been tested, but the example in Figure 1 shows variety in both sentence structure and sentence length.

2.4. Word Usage

The word usage measures are an attempt to identify some other constructional features of writing style. There are many different ways in English to say the same thing. The constructions differ from one another in the form of the words used. The following sentences all convey approximately the same meaning but differ in word usage:

- The cxio program is used to perform all communication between the systems.
- The cxio program performs all communications between the systems.
- The cxio program is used to communicate between the systems.
- The cxio program communicates between the systems.
- All communication between the systems is performed by the cxio program.

The distribution of the parts of speech and verb constructions helps identify overuse of particular constructions. Although the measures used by STYLE are crude, they do point out problem areas. For each category, STYLE reports a percentage and a raw count. In addition to looking at the percentage, the user may find it useful to compare the raw count with the number of sentences. If, for example, the number of infinitives is almost equal to the number of sentences, then many of the sentences in the document are constructed like the first and third in the preceding example. The user may want to transform some of these sentences into another form. Some of the implications of the word usage measures are discussed below.

Verbs are measured in several different ways to try to determine what types of verb constructions are most frequent in the document. Technical writing tends to contain many passive verb constructions and other usage of the verb "to be". The category of verbs labeled "tobe" measures both passives and sentences of the form:

subject tobe predicate

In counting verbs, whole verb phrases are counted as one verb. Verb phrases containing auxiliary verbs are counted in the category "aux". The verb phrases counted here are those whose tense is not simple present or simple past. It might eventually be useful to do more detailed measures of verb tense or mood. Infinitives are listed as "inf". The percentages reported for these three categories are based on the total number of verb phrases found. These categories are not mutually exclusive; they cannot be added, since, for example, "to be going" counts as both "tobe" and "inf". Use of these three types of verb constructions varies significantly among authors.

STYLE reports passive verbs as a percentage of the finite verbs in the document. Most style books warn against the overuse of passive verbs. Coleman [11] has shown that sentences with active verbs are easier to learn than those with passive verbs. Although the inverted object-subject order of the passive voice seems to emphasize the object, Coleman's experiments showed that there is little difference in retention by word position. He also showed that the direct object of an active verb is retained better than the subject of a passive verb. These experiments

There are three streets used by the traffic.
 There are too many users on this system.

This construction tends to emphasize the object rather than the subject of the sentence. The flag “-e” will cause STYLE to print all sentences that begin with an expletive.

3. DICTION

The program DICTION prints all sentences in a document containing phrases that are either frequently misused or indicate wordiness. The program, an extension of Aho’s FGREP [12] string matching program, takes as input a file of phrases or patterns to be matched and a file of text to be searched. A data base of about 450 phrases has been compiled as a default pattern file for DICTION. Before attempting to locate phrases, the program maps upper case letters to lower case and substitutes blanks for punctuation. Sentence boundaries were deemed less critical in DICTION than in STYLE, so abbreviations and other uses of the character “.” are not treated specially. DICTION brackets all pattern matches in a sentence with the characters “[” “]” . Although many of the phrases in the default data base are correct in some contexts, in others they indicate wordiness. Some examples of the phrases and suggested alternatives are:

Phrase	Alternative
a large number of	many
arrive at a decision	decide
collect together	collect
for this reason	so
pertaining to	about
through the use of	by or with
utilize	use
with the exception of	except

Appendix 2 contains a complete list of the default file. Some of the entries are short forms of problem phrases. For example, the phrase “the fact” is found in all of the following and is sufficient to point out the wordiness to the user:

Phrase	Alternative
accounted for by the fact that	caused by
an example of this is the fact that	thus
based on the fact that	because
despite the fact that	although
due to the fact that	because
in light of the fact that	because
in view of the fact that	since
notwithstanding the fact that	although

Entries in Appendix 2 preceded by “” are not matched. See Section 7 for details on the use of “”.

The user may supply her/his own pattern file with the flag “-f patfile”. In this case the default file will be loaded first, followed by the user file. This mechanism allows users to suppress patterns contained in the default file or to include their own pet peeves that are not in the default file. The flag “-n” will exclude the default file altogether. In constructing a pattern file, blanks should be used before and after each phrase to avoid matching substrings in words. For example, to find all occurrences of the word “the”, the pattern “ the ” should be used. The blanks cause only the word “the” to be matched and not the string “the” in words like there, other, and therefore. One side effect of surrounding the words with blanks is that when two phrases occur without intervening words, only the first will be matched.

types of sentences, while author 2 prefers simple and complex sentences, using few compound or compound-complex sentences. The other major difference in the styles of these authors is the location of subordination. Author 1 seems to prefer embedded or trailing subordination, while author 2 begins many sentences with the subordinate clause. The documents tested for both authors 1 and 2 were technical documents, written for a technical audience. The instructional documents, which are written for craftspeople, vary surprisingly little from the two technical samples. The sentences and words are a little longer, and they contain many passive and auxiliary verbs, few adverbs, and almost no pronouns. The instructional documents contain many imperative sentences, so there are many sentence with verb openers. The sample of Federalist Papers contrasts with the other samples in almost every way.

Table 2
Text Statistics on Single Authors

	variable	author 1	author 2	inst.	FED
readability	Kincaid	11.0	10.3	10.8	16.3
	automated	11.0	10.3	11.9	17.8
	Coleman-Liau	9.3	10.1	10.2	12.3
	Flesch	10.3	10.7	10.1	15.0
sentence info	av sent length	22.64	19.61	22.78	31.85
	av word length	4.47	4.66	4.65	4.95
	av nonfunction length	5.64	5.92	6.04	6.87
	short sent	35%	43%	35%	40%
	long sent	18%	15%	16%	21%
sentence types	simple	36%	43%	40%	31%
	complex	34%	41%	37%	34%
	compound	13%	7%	4%	10%
	compound-complex	16%	8%	14%	25%
verb type	tobe	42%	43%	45%	37%
	auxiliary	17%	19%	32%	32%
	infinitives	17%	15%	12%	21%
	passives	20%	19%	36%	20%
word usage	prepositions	10.0%	10.8%	12.3%	15.9%
	conjunctions	3.2%	2.4%	3.9%	3.4%
	adverbs	5.05%	4.6%	3.5%	3.7%
	nouns	27.7%	26.5%	29.1%	24.9%
	adjectives	17.0%	19.0%	15.4%	12.4%
	pronouns	5.3%	4.3%	2.1%	6.5%
	nominalizations	1%	2%	2%	3%
sentence openers	prepositions	11%	14%	6%	5%
	adverbs	9%	9%	6%	4%
	subject	65%	59%	54%	66%
	verb	3%	2%	14%	2%
	subordinating conj	8%	14%	11%	3%
	conjunction	1%	0%	0%	3%
	expletives	3%	3%	0%	3%

5.2. DICTION

In the few weeks that DICTION has been available to users about 35,000 sentences have been run with about 5,000 string matches. The authors using the program seem to make the suggested changes about 50-75% of the time. To date, almost 200 of the 450 strings in the default file have been matched. Although most of these phrases are valid and correct in some contexts, the 50-75% change rate seems to show that the phrases are used much more often than concise diction warrants.

words. For this reason, header files containing the definition of the EQN delimiters must also be included as input to STYLE. End markers are often hidden when an equation ends a sentence and the period is typed inside the EQN delimiters.

5. Add a "." after lists. If the flag `-ml` is also used, all lists are suppressed. This is a separate flag because of the variety of ways the list macros are used. Often, lists are sentences that should be included in the analysis. The user must determine how lists are used in the document to be analyzed.

Both STYLE and DICTION call DEROFF before they look at the text. The user should supply the `-ml` flag if the document contains many lists of non-sentences that should be skipped.

7.2. Details of DICTION

The program DICTION is based on the string matching program FGREP. FGREP takes as input a file of patterns to be matched and a file to be searched and outputs each line that contains any of the patterns with no indication of which pattern was matched. The following changes have been added to FGREP:

1. The basic unit that DICTION operates on is a sentence rather than a line. Each sentence that contains one of the patterns is output.
2. Upper case letters are mapped to lower case.
3. Punctuation is replaced by blanks.
4. All pattern matches in the sentence are found and surrounded with "[" "]" .
5. A method for suppressing a string match has been added. Any pattern that begins with "" will not be matched. Because the matching algorithm finds the longest substring, the suppression of a match allows words in some correct contexts not to be matched while allowing the word in another context to be found. For example, the word "which" is often incorrectly used instead of "that" in restrictive clauses. However, "which" is usually correct when preceded by a preposition or ",". The default pattern file suppresses the match of the common prepositions or a double blank followed by "which" and therefore matches only the suspect uses. The double blank accounts for the replaced comma.

8. Conclusions

A system of writing tools that measure some of the objective characteristics of writing style has been developed. The tools are sufficiently general that they may be applied to documents on any subject with equal accuracy. Although the measurements are only of the surface structure of the text, they do point out problem areas. In addition to helping writers produce better documents, these programs may be useful for studying the writing process and finding other formulae for measuring readability.

Appendix 1

STYLE Abbreviations

a. d.
A. M.
a. m.
b. c.
Ch.
ch.
ckts.
dB.
Dept.
dept.
Depts.
depts.
Dr.
Drs.
e. g.
Eq.
eq.
et al.
etc.
Fig.
fig.
Figs.
figs.
ft.
i. e.
in.
Inc.
Jr.
jr.
mi.
Mr.
Mrs.
Ms.
No.
no.
Nos.
nos.
P. M.
p. m.
Ph. D.
Ph. d.
Ref.
ref.
Refs.
refs.
St.
vs.
yr.

more preferable	seems apparent	worth while
most unique	send a communication	would of
must of	short space of time	ing behavior
mutual cooperation	should of	wise
necessary requisite	single unit	~ which
necessitate	situation	~ about which
need for	so as to	~ after which
nice	sort of	~ at which
not be un	spell out	~ between which
not in a position to	still continue	~ by which
not of a high order of accuracy	still remain	~ for which
not un	subsequent	~ from which
notwithstanding	substantially in agreement	~ in which
of considerable magnitude	succeed in	~ into which
of that	suggestive of	~ of which
of the opinion that	superior than	~ on which
off of	surrounding circumstances	~ on which
on a few occasions	take appropriate	~ over which
on account of	take cognizance of	~ through which
on behalf of	take into consideration	~ to which
on the grounds that	termed as	~ under which
on the occasion	terminate	~ upon which
on the part of	termination	~ with which
one of the	the author	~ without which
open up	the authors	"clockwise
operates to correct	the case that	"likewise
outside of	the fact	"otherwise
over with	the foregoing	
overall	the foreseeable future	
past history	the fullest possible extent	
perceptive of	the majority of	
perform a measurement	the nature	
perform the measurement	the necessity of	
permits the reduction of	the only difference being that	
personalize	the order of	
pertaining to	the point that	
physical size	the truth is	
plan ahead	there are not many	
plan for the future	through the medium of	
plan in advance	through the use of	
plan on	throughout the entire	
present a conclusion	time interval	
present a report	to summarize the above	
presently	total effect of all this	
prior to	totality	
prioritize	transpire	
proceed to	true facts	
procure	try and	
productive of	ultimate end	
prolong the duration	under a separate cover	
protrude out from	under date of	
provided that	under separate cover	
pursuant to	under the necessity to	
put to use in	underlying purpose	
range all the way from	undertake a study	
reason is because	uniformly consistent	
reason why	unique	
recur again	until such time as	
reduce down	up to this time	
refer back	upshot	
reference to this	utilize	
reflective of	very	
regarding	very complete	
regretful	very unique	
reinitiate	vital	
relative to	which	
repeat again	with a view to	
representative of	with reference to	
resultant effect	with regard to	
resume again	with the exception of	
retreat back	with the object of	
return again	with the result that	
return back	with this in mind, it is clear that	
revert back	within the realm of possibility	
seal off	without further delay	

A Guide to the Dungeons of Doom

*Michael C. Toy
Kenneth C. R. C. Arnold*

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

ABSTRACT

Rogue is a visual CRT based fantasy game which runs under the UNIX† timesharing system. This paper describes how to play rogue, and gives a few hints for those who might otherwise get lost in the Dungeons of Doom.

1. Introduction

You have just finished your years as a student at the local fighter's guild. After much practice and sweat you have finally completed your training and are ready to embark upon a perilous adventure. As a test of your skills, the local guildmasters have sent you into the Dungeons of Doom. Your task is to return with the Amulet of Yendor. Your reward for the completion of this task will be a full membership in the local guild. In addition, you are allowed to keep all the loot you bring back from the dungeons.

In preparation for your journey, you are given an enchanted mace, a bow, and a quiver of arrows taken from a dragon's hoard in the far off Dark Mountains. You are also outfitted with elf-crafted armor and given enough food to reach the dungeons. You say goodbye to family and friends for what may be the last time and head up the road.

You set out on your way to the dungeons and after several days of uneventful travel, you see the ancient ruins that mark the entrance to the Dungeons of Doom. It is late at night, so you make camp at the entrance and spend the night sleeping under the open skies. In the morning you gather your weapons, put on your armor, eat what is almost your last food, and enter the dungeons.

2. What is going on here?

You have just begun a game of rogue. Your goal is to grab as much treasure as you can, find the Amulet of Yendor, and get out of the Dungeons of Doom alive. On the screen, a map of where you have been and what you have seen on the current dungeon level is kept. As you explore more of the level, it appears on the screen in front of you.

Rogue differs from most computer fantasy games in that it is screen oriented. Com-

†UNIX is a trademark of Bell Laboratories

- Arm** Your current armor protection. This number indicates how effective your armor is in stopping blows from unfriendly creatures. The higher this number is, the more effective the armor.
- Exp** These two numbers give your current experience level and experience points. As you do things, you gain experience points. At certain experience point totals, you gain an experience level. The more experienced you are, the better you are able to fight and to withstand magical attacks.

3.2. The top line

The top line of the screen is reserved for printing messages that describe things that are impossible to represent visually. If you see a "--More--" on the top line, this means that rogue wants to print another message on the screen, but it wants to make certain that you have read the one that is there first. To read the next message, just type a space.

3.3. The rest of the screen

The rest of the screen is the map of the level as you have explored it so far. Each symbol on the screen represents something. Here is a list of what the various symbols mean:

- @ This symbol represents you, the adventurer.
- | These symbols represent the walls of rooms.
- + A door to/from a room.
- . The floor of a room.
- # The floor of a passage between rooms.
- * A pile or pot of gold.
-) A weapon of some sort.
-] A piece of armor.
- ! A flask containing a magic potion.
- ? A piece of paper, usually a magic scroll.
- = A ring with magic properties
- / A magical staff or wand
- ^ A trap, watch out for these.
- % A staircase to other levels
- : A piece of food.

A-Z The uppercase letters represent the various inhabitants of the Dungeons of Doom. Watch out, they can be nasty and vicious.

4. Commands

Commands are given to rogue by typing one or two characters. Most commands can be preceded by a count to repeat them (e.g. typing "10s" will do ten searches). Commands for which counts make no sense have the count ignored. To cancel a count or a prefix, type <ESCAPE>. The list of commands is rather long, but it can be read at any time during the game with the "?" command. Here it is for reference, with a short explanation of each command.

- ? The help command. Asks for a character to give help on. If you type a "***", it will list all the commands, otherwise it will explain what the character you typed does.

- w Wield a weapon. Take a weapon out of your pack and carry it for use in combat, replacing the one you are currently using (if any).
 - W Wear armor. You can only wear one suit of armor at a time. This takes extra time.
 - T Take armor off. You can't remove armor that is cursed. This takes extra time.
 - P Put on a ring. You can wear only two rings at a time (one on each hand). If you aren't wearing any rings, this command will ask you which hand you want to wear it on, otherwise, it will place it on the unused hand. The program assumes that you wield your sword in your right hand.
 - R Remove a ring. If you are only wearing one ring, this command takes it off. If you are wearing two, it will ask you which one you wish to remove,
 - d Drop an object. Take something out of your pack and leave it lying on the floor. Only one object can occupy each space. You cannot drop a cursed object at all if you are wielding or wearing it.
 - c Call an object something. If you have a type of object in your pack which you wish to remember something about, you can use the call command to give a name to that type of object. This is usually used when you figure out what a potion, scroll, ring, or staff is after you pick it up, or when you want to remember which of those swords in your pack you were wielding.
 - D Print out which things you've discovered something about. This command will ask you what type of thing you are interested in. If you type the character for a given type of object (e.g. "!" for potion) it will tell you which kinds of that type of object you've discovered (i.e., figured out what they are). This command works for potions, scrolls, rings, and staves and wands.
 - o Examine and set options. This command is further explained in the section on options.
 - ^R Redraws the screen. Useful if spurious messages or transmission errors have messed up the display.
 - ^P Print last message. Useful when a message disappears before you can read it. This only repeats the last message that was not a mistyped command so that you don't loose anything by accidentally typing the wrong character instead of ^P.
- <ESCAPE>
 Cancel a command, prefix, or count.
- ! Escape to a shell for some commands.
 - Q Quit. Leave the game.
 - S Save the current game in a file. It will ask you whether you wish to use the default save file. *Caveat:* Rogue won't let you start up a copy of a saved game, and it removes the save file as soon as you start up a restored game. This is to prevent people from saving a game just before a dangerous position and then restarting it if they die. To restore a saved game, give the file name as an argument to rogue. As in
 % rogue *save_file*
 To restart from the default save file (see below), run
 % rogue -r
 - v Prints the program version number.
 -) Print the weapon you are currently wielding
 -] Print the armor you are currently wearing
 - = Print the rings you are currently wearing

<i>Type</i>	<i>Protection</i>
None	0
Leather armor	2
Studded leather / Ring mail	3
Scale mail	4
Chain mail	5
Banded mail / Splint mail	6
Plate mail	7

If a piece of armor is enchanted, its armor protection will be higher than normal. If a suit of armor is cursed, its armor protection will be lower, and you will not be able to remove it. However, not all armor with a protection that is lower than normal is cursed.

The commands to use weapons are "W" (wear) and "T" (take off).

7.3. Scrolls

Scrolls come with titles in an unknown tongue³. After you read a scroll, it disappears from your pack. The command to use a scroll is "r" (read).

7.4. Potions

Potions are labeled by the color of the liquid inside the flask. They disappear after being quaffed. The command to use a scroll is "q" (quaff).

7.5. Staves and Wands

Staves and wands do the same kinds of things. Staves are identified by a type of wood; wands by a type of metal or bone. They are generally things you want to do to something over a long distance, so you must point them at what you wish to affect to use them. Some staves are not affected by the direction they are pointed, though. Staves come with multiple magic charges, the number being random, and when they are used up, the staff is just a piece of wood or metal.

The command to use a wand or staff is "z" (zap)

7.6. Rings

Rings are very useful items, since they are relatively permanent magic, unlike the usually fleeting effects of potions, scrolls, and staves. Of course, the bad rings are also more powerful. Most rings also cause you to use up food more rapidly, the rate varying with the type of ring. Rings are differentiated by their stone settings. The commands to use rings are "P" (put on) and "R" (remove).

7.7. Food

Food is necessary to keep you going. If you go too long without eating you will faint, and eventually die of starvation. The command to use food is "e" (eat).

8. Options

Due to variations in personal tastes and conceptions of the way rogue should do things, there are a set of options you can set that cause rogue to behave in various different ways.

³ Actually, it's a dialect spoken only by the twenty-seven members of a tribe in Outer Mongolia, but you're not supposed to know that.

tombstone [*tombstone*]

Print out the tombstone at the end if you get killed. This is nice but slow, so you can turn it off if you like.

inven [*overwrite*]

Inventory type. This can have one of three values: *overwrite*, *slow*, or *clear*. With *overwrite* the top lines of the map are overwritten with the list when inventory is requested or when "Which item do you wish to . . . ?" questions are answered with a "***". However, if the list is longer than a screenful, the screen is cleared. With *slow*, lists are displayed one item at a time on the top of the screen, and with *clear*, the screen is cleared, the list is displayed, and then the dungeon level is re-displayed. Due to speed considerations, *clear* is the default for terminals without clear-to-end-of-line capabilities.

name [*account name*]

This is the name of your character. It is used if you get on the top ten scorer's list.

fruit [*slime-mold*]

This should hold the name of a fruit that you enjoy eating. It is basically a whimsey that rogue uses in a couple of places.

file [*~/rogue.save*]

The default file name for saving the game. If your phone is hung up by accident, rogue will automatically save the game in this file. The file name may start with the special character "~" which expands to be your home directory.

9. Scoring

Rogue usually maintains a list of the top scoring people or scores on your machine. Depending on how it is set up, it can post either the top scores or the top players. In the latter case, each account on the machine can post only one non-winning score on this list. If you score higher than someone else on this list, or better your previous score on the list, you will be inserted in the proper place under your current name. How many scores are kept can also be set up by whoever installs it on your machine.

If you quit the game, you get out with all of your gold intact. If, however, you get killed in the Dungeons of Doom, your body is forwarded to your next-of-kin, along with 90% of your gold; ten percent of your gold is kept by the Dungeons' wizard as a fee⁶. This should make you consider whether you want to take one last hit at that monster and possibly live, or quit and thus stop with whatever you have. If you quit, you do get all your gold, but if you swing and live, you might find more.

If you just want to see what the current top players/games list is, you can type
% rogue -s

10. Acknowledgements

Rogue was originally conceived of by Glenn Wichman and Michael Toy. Ken Arnold and Michael Toy then smoothed out the user interface, and added jillions of new features. We would like to thank Bob Arnold, Michelle Busch, Andy Hatcher, Kipp Hickman, Mark Horton, Daniel Jensen, Bill Joy, Joe Kalash, Steve Maurer, Marty McNary, Jan Miller, and Scott Nelson for their ideas and assistance; and also the teeming multitudes who graciously ignored work, school, and social life to play rogue and send us bugs, complaints, suggestions, and just plain flames. And also Mom.

⁶ The Dungeon's wizard is named Wally the Wonder Badger. Invocations should be accompanied by a sizable donation.

STAR TREK

by

Eric Allman
University of California
Berkeley

INTRODUCTION

Well, the federation is once again at war with the Klingon empire. It is up to you, as captain of the U.S.S. Enterprise, to wipe out the invasion fleet and save the Federation.

For the purposes of the game the galaxy is divided into 64 quadrants on an eight by eight grid, with quadrant 0,0 in the upper left hand corner. Each quadrant is divided into 100 sectors on a ten by ten grid. Each sector contains one object (e.g., the Enterprise, a Klingon, or a star).

Navigation is handled in degrees, with zero being straight up and ninety being to the right. Distances are measured in quadrants. One tenth quadrant is one sector.

The galaxy contains starbases, at which you can dock to refuel, repair damages, etc. The galaxy also contains stars. Stars usually have a knack for getting in your way, but they can be triggered into going nova by shooting a photon torpedo at one, thereby (hopefully) destroying any adjacent Klingons. This is not a good practice however, because you are penalized for destroying stars. Also, a star will sometimes go supernova, which obliterates an entire quadrant. You must never stop in a supernova quadrant, although you may "jump over" one.

Some starsystems have inhabited planets. Klingons can attack inhabited planets and enslave the populace, which they then put to work building more Klingon battle cruisers.

STARTING UP THE GAME

To request the game, issue the command

```
/usr/games/trek
```

from the shell. If a filename is supplied, a log of the game is written onto that file. (Otherwise, no file is written.) If the "-a" flag is stated before the filename, the log of the game is

Mnemonic: lrsca
 Shortest Abbreviation: l
 Consumes: nothing



Long range scan gives you information about the eight quadrants that surround the quadrant you're in. A sample long range scan follows:

Long range scan for quadrant 0,3

	2	3	4	
	!	*	!	*
	!	*	!	*
0	!	108	!	6
1	!	9	!	///
	!	19	!	8

The three digit numbers tell the number of objects in the quadrants. The units digit tells the number of stars, the tens digit the number of starbases, and the hundreds digit is the number of Klingons. "*" indicates the negative energy barrier at the edge of the galaxy, which you cannot enter. "///" means that that is a supernova quadrant and must not be entered.

Damage Report

Mnemonic: damages
 Shortest Abbreviation: da
 Consumes: nothing

A damage report tells you what devices are damaged and how long it will take to repair them. Repairs proceed faster when you are docked at a starbase.

Set Warp Factor

Mnemonic: warp
 Shortest Abbreviation: w
 Full Command: warp factor
 Consumes: nothing

The warp factor tells the speed of your starship when you move under warp power (with the *move* command). The higher the warp factor, the faster you go, and the more energy you use.

The minimum warp factor is 1.0 and the maximum is 10.0. At speeds above warp 6 there is danger of the warp engines being damaged. The probability of this increases at higher warp speeds. Above warp 9.0 there is a chance of entering a time warp.

Move Under Warp Power

Shields protect you from Klingon attack and nearby novas. As they protect you, they weaken. A shield which is 78% effective will absorb 78% of a hit and let 22% in to hurt you.

The Klingons have a chance to attack you every time you raise or lower shields. Shields do not rise and lower instantaneously, so the hit you receive will be computed with the shields at an intermediate effectiveness.

It takes energy to raise shields, but not to drop them.

Cloaking Device

Mnemonic: cloak
Shortest Abbreviation: cl
Full Command: cloak up/down
Consumes: energy

When you are cloaked, Klingons cannot see you, and hence they do not fire at you. They are useful for entering a quadrant and selecting a good position, however, weapons cannot be fired through the cloak due to the huge energy drain that it requires.

The cloak up command only starts the cloaking process; Klingons will continue to fire at you until you do something which consumes time.

Fire Phasers

Mnemonic: phasers
Shortest Abbreviation: p
Full Commands: phasers automatic amount
 phasers manual amt1 course1 spread1 ...
Consumes: energy

Phasers are energy weapons; the energy comes from your ship's reserves ("total energy" on a srscon). It takes about 250 units of hits to kill a Klingon. Hits are cumulative as long as you stay in the quadrant.

Phasers become less effective the further from a Klingon you are. Adjacent Klingons receive about 90% of what you fire, at five sectors about 60%, and at ten sectors about 35%. They have no effect outside of the quadrant.

Phasers cannot be fired while shields are up; to do so would fry you. They have no effect on starbases or stars.

In automatic mode the computer decides how to divide up the energy among the Klingons present; in manual mode you do that yourself.

trajectory -- prints the course and distance to all the Klingons in the quadrant.

warpcost dist warp_factor -- computes the cost in time and energy to move 'dist' quadrants at warp 'warp_factor'.

impcost dist -- same as warpcost for impulse engines.

pheff range -- tells how effective your phasers are at a given range.

distresslist -- gives a list of currently distressed starbases and starsystems.

More than one request may be stated on a line by separating them with semi-colons.

Dock at Starbase

Mnemonic: dock
Shortest Abbreviation: do
Consumes: nothing

You may dock at a starbase when you are in one of the eight adjacent sectors.

When you dock you are resupplied with energy, photon torpedoes, and life support reserves. Repairs are also done faster at starbase. Any prisoners you have taken are unloaded. You do not receive points for taking prisoners until this time.

Starbases have their own deflector shields, so you are safe from attack while docked.

Undock from Starbase

Mnemonic: undock
Shortest Abbreviation: u
Consumes: nothing

This just allows you to leave starbase so that you may proceed on your way.

Rest

Mnemonic: rest
Shortest Abbreviation: r
Full Command: rest time
Consumes: time

This command allows you to rest to repair damages. It is not advisable to rest while under attack.

and there is an inhabitable starsystem in the area, the crew beams down, otherwise you leave them to die. You are given an old but still usable ship, the Faire Queene.

Ram

Mnemonic: ram
Shortest Abbreviation: ram
Full Command: ram course distance
Consumes: time and energy

This command is identical to "move", except that the computer doesn't stop you from making navigation errors.

You get very nearly slaughtered if you ram anything.

Self Destruct

Mnemonic: destruct
Shortest Abbreviation: destruct
Consumes: everything

Your starship is self-destructed. Chances are you will destroy any Klingons (and stars, and starbases) left in your quadrant.

Terminate the Game

Mnemonic: terminate
Shortest Abbreviation: terminate
Full Command: terminate yes/no

Cancels the current game. No score is computed. If you answer yes, a new game will be started, otherwise trek exits.

Call the Shell

Mnemonic: shell
Shortest Abbreviation: shell

Temporarily escapes to the shell. When you exit the shell you will return to the game.

SCORING

The scoring algorithm is rather complicated. Basically, you get points for each Klingon you kill, for your Klingon per stardate kill rate, and a bonus if you win the game. You lose points for the number of Klingons left in the galaxy at the end of the game, for getting killed, for each star, starbase, or inhabited starsystem you



NOTES

NOTES

UNIX Documents

URM	User Reference Manual	PS1:6	Berkeley Software Architecture Manual (4.3 Edition)
	man section 1 (commands)	PS1:7	Introductory 4.3BSD Interprocess Communication
	man section 6 (games)	PS1:8	Advanced 4.3BSD Interprocess Communication
	man section 7 (miscellaneous)	PS1:9	Lint, A C Program Checker
USD	User Supplementary Documents	PS1:10	ADB Tutorial
USD:1	Unix for Beginners	PS1:11	Debugging with dbx
USD:2	Learn – Computer–Aided Instruction	PS1:12	Make
USD:3	Introduction to the UNIX Shell	PS1:13	Revision Control System (RCS)
USD:4	Introduction to the C shell	PS1:14	Source Code Control System (SCCS)
USD:5	DC – Interactive Desk Calculator	PS1:15	YACC: Yet Another Compiler-Compiler
USD:6	BC – Arbitrary Precision Desk-Calculator	PS1:16	LEX – A Lexical Analyzer Generator
USD:7	Mail Reference Manual	PS1:17	M4 Macro Processor
USD:8	MH Message Handling System	PS1:18	curses library
USD:9	How to Read the Network News		
USD:10	How to Use USENET Effectively	PS2	Programmer Supplementary Documents, part 2
USD:11	Notesfile Reference Manual	PS2:1	The Unix Time–Sharing System
USD:12	Tutorial Introduction to “ed”	PS2:2	UNIX 32/V – Summary
USD:13	Advanced Editing on Unix	PS2:3	Unix Programming – Second Edition
USD:14	Edit: A Tutorial	PS2:4	Unix Implementation
USD:15	Introduction to Display Editing with Vi	PS2:5	The Unix I/O System
USD:16	Ex Reference Manual (Version 3.7)	PS2:6	Programming Language EFL
USD:17	Jove Manual for UNIX Users	PS2:7	Berkeley FP User’s Manual
USD:18	SED – A Non-interactive Text Editor	PS2:8	Ratfor – Preprocessor for Rational FORTRAN
USD:19	AWK – Pattern Scanning/Processing Language	PS2:9	The FRANZ LISP Manual
USD:20	Using the –ms Macros with Troff and Nroff	PS2:10	Ingres (Version 8) Reference Manual
USD:21	Revised Version of –ms		
USD:22	Writing Papers with <i>nroff</i> using –me	SMM	System Manager’s Manual
USD:23	–me Reference Manual		man section 8 (system administration)
USD:24	NROFF/TROFF User’s Manual	SMM:1	Installing and Operating 4.3BSD
USD:25	TROFF Tutorial	SMM:2	Building 4.3BSD Systems with <i>Config</i>
USD:26	Typesetting Mathematics (eqn)	SMM:3	Using ADB to Debug the Kernel
USD:27	Typesetting Mathematics – User’s Guide	SMM:4	Disc Quotas
USD:28	Tbl – A Program to Format Tables	SMM:5	Fscck – File System Check Program
USD:29	Refer – A Bibliography System	SMM:6	Line Printer Spooler Manual
USD:30	Some Applications of Inverted Indexes ...	SMM:7	Sendmail Installation and Operation
USD:31	BIB – Bibliography Formatting Program	SMM:8	Timed Installation and Operation
USD:32	Writing Tools – STYLE and DICTION	SMM:9	UUCP Implementation Description
USD:33	A Guide to the Dungeons of Doom	SMM:10	USENET Version B Installation
USD:34	Star Trek	SMM:11	Name Server Operations Guide
		SMM:12	Bug Fixes and Changes in 4.3BSD
		SMM:13	Changes to the Kernel in 4.3BSD
PRM	Programmer Reference Manual	SMM:14	A Fast File System for UNIX
	man sections 2 (system calls)	SMM:15	4.3BSD Networking Implementation Notes
	man sections 3 (library routines)	SMM:16	Sendmail – An Internetwork Mail Router
	man sections 4 (devices, special files)	SMM:17	On the Security of UNIX
	man sections 5 (file formats)	SMM:18	Password Security – A Case History
PS1	Programmer Supplementary Docs, part 1	SMM:19	A Tour Through the Portable C Compiler
PS1:1	C Language – Reference Manual	SMM:20	Writing NROFF Terminal Descriptions
PS1:2	Fortran 77	SMM:21	A Dial–Up Network of UNIX Systems
PS1:3	f77 I/O Library	SMM:22	Berkeley Time Synchronization Protocol
PS1:4	Berkeley Pascal User’s Manual		
PS1:5	Vax Assembler Reference Manual		