ALTOS
COMPUTER SYSTEMS

ALTOS 586 Computer System

XENIX Development System Programmer's Reference Guide

Part Number: 690-13128-001

January 1983

ACKNOWLEDGEMENTS

Altos is a registered trademark of Altos Computer Systems.

XENIX is a trademark of Microsoft, Incorporated and is a 16-bit microcomputer implementation of the UNIX operating system, version 7.

UNIX is a trademark of Bell Laboratories

UNET is a trademark of 3Com Corporation

The information contained herein is subject to change without notice. Changes will be incorporated in new editions of the document as they are published.

TABLE OF CONTENTS

H. 8086 ASSEMBLY LANGUAGE REFERENCE MANUAL

XENIX Software Development Extract from Microsoft Manual

I. TUTORIAL AND REFERENCE MATERIAL (UNIVERSITY OF CALIFORNIA, BERKELEY MANUALS)

An Introduction to the C Shell
An Introduction to Display Editing with Vi
Quick Reference for Ex,Vi
Ex Reference Manual
Edit: A Tutorial
Ex/Edit Command Summary (Version 2.0)
Mail Reference Manual
-ME Reference Manual
Screen Updating and Cursor Movement Organization
Screen Updating and Cursor Movement Organization: A Library Package

Appendix B: Floppy Diskette Organization

A brief reference describing how files are allocated on floppy diskettes.

Appendix C. The Serial Line Printer and Spooler

Information regarding the serial line printer and spooler, such as hardware connections required, configuring your system without a printer, connecting more than one printer, and changing/setting band rates.

Appendix D. List of Terminal Capabilities

A data base listing special capabilities of all terminals supported by Altos XENIX.

Appendix E. Numeric Formats, C, and Fortran 77

Reference information on the internal format used for numerical representation in these languages.

Appendix F. Sample List of XENIX Utilities

A sample list of utilities furnished with your system.

Appendix G. Copying files from the Altos 8600 to the Altos 586 under the XENIX operating system.

A description on how to transfer files from an Altos 8600 to an Altos 586 computer system under the XENIX operating system, or between two 586 computer systems. It discusses the uucp (UNIX-to-UNIX copy) Facility.

Appendix H. 8086 Assembly Language

A description of the XENIX 8086 Assembly Language.

Appendix I. Tutorial and Reference Material (University of California, Berkeley Manuals)

Documentation describing UNIX modifications developed at the University of California, Berkeley. The material is supplied from the Regents of the University.

Appendix B: Floppy Diskette Organization

A brief reference describing how files are allocated on floppy diskettes.

Appendix C. The Serial Line Printer and Spooler

Information regarding the serial line printer and spooler, such as hardware connections required, configuring your system without a printer, connecting more than one printer, and changing/setting band rates.

Appendix D. List of Terminal Capabilities

A data base listing special capabilities of all terminals supported by Altos XENIX.

Appendix E. Numeric Formats, C, and Fortran 77

Reference information on the internal format used for numerical representation in these languages.

Appendix F. Sample List of XENIX Utilities

A sample list of utilities furnished with your system.

Appendix G. Copying files from the Altos 8600 to the Altos 586 under the XENIX operating system.

A description on how to transfer files from an Altos 8600 to an Altos 586 computer system under the XENIX operating system, or between two 586 computer systems. It discusses the uucp (UNIX-to-UNIX copy) Facility.

Appendix H. 8086 Assembly Language

A description of the XENIX 8086 Assembly Language.

Appendix I. Tutorial and Reference Material (University of California, Berkeley Manuals)

Documentation describing UNIX modifications developed at the University of California, Berkeley. The material is supplied from the Regents of the University.

information on Unix programming (C language).

Volume 2B contains additional reference material, and includes advanced topics and languages. For example, this volume includes information or supporting tools and languages such as yacc, which is a tool for writing compilers for other languages. It also includes information on system implementation and maintenance.

UNIX Reference Card

A 36 page concise reference booklet, loosely bound in order to lie flat. It contains information on UNIX commands, documentation preparation, and C language functions.

Commercially Available Books

There are numerous commercially available books on UNIX that explain it and give tutorial material. Two such books are:

A User Guide to the UNIX System, by Thomas and Yates

Using the UNIX System, by Richard Gauthier

Two useful programming books related to UNIX are:

The C Programming Language, by Kernighan and Ritchie.
This book describes the C programming language, which is the language that the UNIX operating system is written in. It provides tutorials as well as a reference section.

Software Tools, by Kernighan and Plauger.
This books is a guide to good programming techniques and a source of proven, useful programs written in RatFor (Rational Fortan). The C language, which is designed for UNIX, provided the model for RatFor. Many of the tools described in this book are based on UNIX models.

information on Unix programming (C language).

Volume 2B contains additional reference material, and includes advanced topics and languages. For example, this volume includes information or supporting tools and languages such as yacc, which is a tool for writing compilers for other languages. It also includes information on system implementation and maintenance.

UNIX Reference Card

A 36 page concise reference booklet, loosely bound in order to lie flat. It contains information on UNIX commands, documentation preparation, and C language functions.

Commercially Available Books

There are numerous commercially available books on UNIX that explain it and give tutorial material. Two such books are:

A User Guide to the UNIX System, by Thomas and Yates

Using the UNIX System, by Richard Gauthier

Two useful programming books related to UNIX are:

The C Programming Language, by Kernighan and Ritchie.
This book describes the C programming language, which is the language that the UNIX operating system is written in. It provides tutorials as well as a reference section.

Software Tools, by Kernighan and Plauger.
This books is a guide to good programming techniques and a source of proven, useful programs written in RatFor (Rational Fortan). The C language, which is designed for UNIX, provided the model for RatFor. Many of the tools described in this book are based on UNIX models.

Section 2

INSTALLING XENIX DEVELOPMENT SYSTEM

To install the Xenix Development System on your Altos 586 Computer System, you should:

1. Install the Xenix Run-Time System by following the instructions in the **Altos Introduction to Xenix Manual.** Do not shut the system down.

   If you interrupt the installation procedure for some reason, or your system was shut down by a power failure or system crash, see the Resuming Interrupted Installation section in the **Altos Introduction to XENIX Manual.**

2. Make sure you are logged as super-user (root).

3. Enter

   cd / <cr>

   This command causes the system to go to the top directory (or parent directory) of the XENIX system.

4. Insert the diskette labeled "Xenix Utilities #2 of n," where "n" is the total number of utility diskettes.

   Enter

   tar xv <cr>

   This command causes the directories and files on the utility diskette to be loaded onto the XENIX system. For information on the tar utility, see the section SAVING AND RESTORING FILES in the **Altos Introduction to XENIX Manual** or under the entries for Tar(1), DD(2), Dump (1) and Restore (1) in the **UNIX Programmer's Manual.**

5. Repeat step 4 for each utility diskette.

6. When you have loaded all of the utility diskettes, enter

   install <cr>

7. To load the C compiler onto the XENIX system, you should:

   Insert the diskette labeled "C Compiler."

   Enter

   tar xv <cr>

   Enter

install

You have just loaded the C Compiler.

8. If you wish to load the UNIX Fortran compiler, you should:

   Insert the diskette labeled "F77."

   Enter

   cd /tmp <cr>

   Enter

   install <cr>

   You have just loaded the UNIX Fortran compiler.

9. If the prior steps were successful, your XENIX Development System is correctly installed.

   If you purchased Altos communication network services, refer to the **Altos 586 UNET User Guide** for information on how to install the communication network services.

   If you purchased the ABS package or other Altos application packages, refer to the **Altos XENIX Application Software User Guide** for information on how to install the ABS-586 Menu Shell and application programs.

   If you wish to start a XENIX up, see the "Start-Up XENIX" section of the **Altos Introduction to XENIX Manual.** If the system has not been shutdown, skip steps 2 and 3.

   If you don't plan on using your XENIX system at this time, you can shut the system down by entering:

   # etc/haltsys <cr>
   ** Normal System Shutdown **

Section 3

UTILITY PROGRAMS REFERENCE GUIDE

USEFUL UTILITIES

Table 3-1 lists some useful utilities that are supplied with the Altos implementation of XENIX. This list is not intended to be complete, but merely a summary of those utilities you will find useful in getting started with XENIX. A complete listing and description for all utilities may be found in the **UNIX Programmer's Manual**, Volume 1.

You may list the full set of utilities supplied with any particular release of XENIX by displaying the contents of the /bin, /usr/bin, and /etc directories. Appendix F contains a sample list of utilities.

The Altos implementation of XENIX provides some utilities which differ from standard UNIX, and also some new utilities from various sources. This section documents the changed and new utilities, as "UNIX Manual Changes and Additions." The material supplied in this section may be kept in this supplement or inserted in the **UNIX Programmer's Manual**, as desired.

In the following pages, "UNIX Manual Changes and Additions," many useful utilities are documented. See Table 3-2 for a quick reference to these utilities. Note in particular: format, fcopy, multiuser, and ua, and the new version of tar. The Business Shell, bsh, has two accompanying utilities, menus and digest.

See also the APPENDIX I for reference and tutorial material on the University of California, Berkeley utilities, such as the screen editior vi.

Table 3-1 **A List of Useful Utilities for Getting Started**

UTILITY	DESCRIPTION
ar	Object library manager and archiver
as	XENIX 8Ø86 relocatable assembler
cat	Display a file
cc	"C" compiler
cd	Change directory. Changes your current position in the File System hierarchy.
chmod	Change mode. Changes file protection attributes
chown	Change file ownership
cmp	Compare two files
cp	Copy a file
diff	Display the differences between two files
ed	The standard UNIX editor
ld	XENIX linkage editor
ls	List. Displays the contents of the current directory
mkdir	Make a new directory
mv	Move. Renames files and directories
od	Displays an octal dump of a file
ps	Display system status
pwd	Print working directory. Displays current position in the directory hierarchy
rm	Remove. Deletes a file
rmdir	Delete a directory
stty	Set terminal options, such as baud rate
tar	File system archiver. May be used for file system dumps and restores

UNIX MANUAL CHANGES AND ADDITIONS

The material in this section may remain in this supplement or be inserted in Sections 1 through 5 of Volume 1 of the **UNIX Programmer's Manual**, as you wish. If you insert these documents into the manual, place them in the sections corresponding to the number in parentheses after the utility name. (Entries within sections are in alphabetic order.)

Some of the utilities are enhancements or variations of existing Bell Laboratories UNIX utilities. Others are completely new.

The origin of each utility is specified (in abbreviated form) in column 2 of Table 3-2.

Utilities labelled "(altos)" are provided by Altos Computer Systems.

Utilities labelled "(bell)" were developed by Bell Laboratories after their current manual was published.

Utilities labelled "(msoft)" were developed by Microsoft, Inc.

Utilities labelled "(uofcb)" were developed at the University of California, Berkeley. They are supplied under license from the Regents of the University.

Table 3-2. List of UNIX Manual Changes and Additions

UTILITY	SOURCE	DESCRIPTION
bsh(1)	(altos)	Business Shell. A menu-driven user system with special guidance and convenience features. It enables you to access the more commonly used UNIX utilities via menus.
csh (1)	(uofcb)	A shell (command interpreter) with C-like syntax.
digest(1)	(altos)	Create menu systems for the Business Shell.
edit (1)	(uofcb)	Text editor (variant of the ex editor for new or casual users).
ex (1)	(uofcb)	Text editor.
fcopy(1)	(altos)	Copy a floppy diskette, while in XENIX.

Table 3-2. List of UNIX Manual Changes and Additions (cont.)

UTILITY	SOURCE	DESCRIPTION
format(1)	(altos)	Format a floppy diskette, while in XENIX.
fsck(1)	(bell)	File system consistency check and interactive repair.
layout(1)	(altos)	Configure a hard disk.
ls (1)	(uofcb)	List contents of directory
Mail (1)	(uofcb)	Send and receive mail. (The U.C.B. "Mail" utility goes in front of, and makes use of, the Bell Labs "mail" utility. The names of the two utilities are distinguished by whether the first letter is capitalized or lower case.)
map(1)	(altos)	Create an alternate sector map for a hard disk drive.
multiuser(1)	(altos)	Bring the system up multiuser.
printenv(1)	(uofcb)	Print out the environment.
ps(1)	(uofcb)	Processor status.
reset(1)	(uofcb)	Reset the terminal status bits to a predefined state.
sizefs(1)	(altos)	Determine the size of a logical device from the layout information associated with a hard disk.
tar(1)	(bell)	Tape or floppy archiver. Dumps and restores hard disk files.
ua(1)	(altos)	User administration. Adds and deletes user accounts on the system.
vi(1)	(uofcb)	Screen oriented (visual) display editor.
locking(2)	(msoft)	Lock or unlock a record of a file.
rdchk(2)	(msoft)	Check if there is data to be read.

Table 3.2 List of UNIX Manual Changes and Additions (Cont.)

UTILITY	SOURCE	DESCRIPTION
curses(3)	(uofcb)	Screen functions with "optional" cursor motion. (Has window capability.)
menus(5)	(altos)	Develop menus for Business Shell.
termcap(5)	(uofcb)	Data base which defines cursor-control sequences for most commonly used CRTs. It is used by most "screen oriented" software, such as the Altos shell and visual screen editor, vi.
ttytype(5)	(altos)	Data base for defining terminal type associated with each 586 serial port.

NAME

bsh -- Altos Computer Systems Business Shell

SYNOPSIS

bsh [-fhas] [menusystem]

DESCRIPTION

Bsh is a menu-driven command language interpreter. It may be installed as the "login shell" in the password file, or it may be invoked directly by the user.

The command is implemented using the termcap and curses facilities from UC Berkeley. It must be run from a terminal which is defined within /etc/termcap.

This command should only be run interactively. A user's terminal may be left in a very strange state if bsh is run in the background.

In the options described below, either "line feed" or "return" performs the newline function.

Options

-f Start bsh in "fast" mode. In this mode, a prompt whose first letter is a lower-case alphabetic or numeric character is executed immediately when the first letter is typed. The system does not wait for a terminating newline. Prompts whose first letter is upper-case alphabetic wait for a terminating newline before executing the requested actions. Fast mode is the default initial mode, if not over-ridden by the command line or the BSHINIT variable (see below). The current mode may be changed during execution through use of the "?mode" command (described below).

-h displays a short help message describing how to invoke bsh.

-z displays a one-line descriptive summary of the syntax used to invoke bsh.

-s Start bsh in "slow" mode. In this mode, all prompts must be terminated by newline before execution occurs. The current mode may be changed during execution through use of the "?mode" command (described below).

A menu system may be specified if desired. In this case, bsh utilizes the designated menu system instead of the default one (/etc/menusys.bin). Prior to use by bsh a menu system must be "digested" using the digest(1) utility. If the specified menu system does not exist or if it is not read-accessible, bsh issues an error message and terminates.

How to create a new menu system and how to update or modify an existing menu system is described in menus(5).

Commands

prompts
Typing any of the prompts on the current menu screen immediately causes the actions associated with the prompt to be executed. It is the responsibility of the menu designer to ensure that reasonable actions exist for each prompt. Selecting a prompt with no associated action causes an error message to be displayed.

An action may be any one of the following:

> Go to a specified menu
> Execute a sh(1) script
> Execute a bsh internal command (e.g. chdir(1))

menuname
Typing the name of a menu causes it to immediately become the current menu. If the menuname is misspelled, or if it does not exist in the current menu system, an error message is displayed.

newline
Typing a newline causes the immediately preceding menu to become the current one. If there is no previous menu, an error message is displayed. Bsh does not distinguish between "line feed" and "return" -- both generate a newline.

? Typing a question mark (?) causes the "help" menu associated with the current menu to be displayed. Help menus are no different from normal menus (except, perhaps, in the type of information they contain). When the current menu is named "xyz", typing a question mark is entirely equivalent to typing "xyz?"

?? Typing a pair of question marks (??) causes the bsh system help information to be displayed. It contains much the same information as is presented here.

menuname?
Typing the name of a menu followed by question mark causes the designated help menu to become the current one.

manualpage??
Typing the name of an entry in the Unix manual followed by two question marks causes the designated manual page to be displayed. Thus, to see the entry for bsh one

may type "bsh??" This is precisely equivalent to typing "!man bsh."

!command
The exclamation point (!) allows the user to "escape" to the standard shell (sh(1)). The command must follow the usual rules as described in the sh(1) documentation. In particular, the command may consist of a sequence of shell commands separated by semicolons -- thus several actions may be invoked. If the command is absent, sh(1) is invoked as a sub-shell with no arguments. In this case, bsh will be resumed as soon as the sub-shell terminates. (Usually, this is accomplished by sending the sub-shell an end-of-file. End-of-file is Control-d on most terminals.) You may escape to the Berkeley C shell (csh(1)) by typing "!csh."

?index
This special command causes bsh to display its internal "index" for the current menu system. The index contains the names of every accessible menu.

?mode
This special command allows the user to change from "slow" mode to "fast" mode and vice versa. The user is asked if he wishes to change to the alternate mode. If your response begins with "y" or "Y", the change is made, otherwise the current mode remains in effect.

interrupt
Bsh will immediately return to the top-level command interpreter upon receipt of an interrupt signal. Such a signal is usually generated via the DEL, RUBOUT or BREAK key.

backspace
Bsh understands the Backspace function (as obtained from /etc/termcap).

CANcel
Bsh interprets the CANcel key to mean "restart input." The CANcel key is Control-x on many of the more popular terminals.

ESCape
Typing an ESCape has the same effect as does typing CANcel.

DC2 If the screen becomes "dirty" for some reason, you can force bsh to clear it and redisplay the current contents by transmitting an ASCII "DC2." This is Control-r on most of the currently popular terminals.

q Typing a "q", "Q" or "Quit" all have the same effect:

bsh is terminated. If bsh is your login shell, "quit" also results in your being logged out.

Environment

BSHINIT

The BSHINIT environment variable contains the initial value of the default mode ("fast" or "slow"). If this variable does not exist in the environment, bsh assumes "fast" mode. BSHINIT should be set by inserting the line BSHINIT="fast" or BSHINIT="slow" into your .profile file.

Note that even if bsh is designated as the "login shell" in /etc/passwd, your .profile file will be interpreted correctly. (See login(1) and sh(1).) In particular, any overriding definitions you may have for the kill and erase characters will be correctly interpreted by bsh.

FILES

~/.profile	contains commands to be executed during login(1)
/etc/menusys.bin	default menu system used by bsh
/etc/passwd	used to define a user's login name, password, home directory, shell, etc.
/etc/termcap	contains terminal attribute descriptions
/usr/lib/bsh.messages	system warning and error messages

SEE ALSO

digest(1M), login(1), menus(5), sh(1), termcap(5)

DIAGNOSTICS

The diagnostics produced by bsh are intended to be self-explanatory.

BUGS

Bsh probably should never allow itself to be run in the background.

Bsh should detect the fact that the current terminal is not defined in /etc/termcap and abort gracefully.

NAME

csh – a shell (command interpreter) with C-like syntax

SYNOPSIS

csh [–cefinstvVxX] [arg ...]

DESCRIPTION

Csh is a command language interpreter. It begins by executing commands from the file '.cshrc' in the *home* directory of the invoker. If this is a login shell then it also executes commands from the file '.login' there. In the normal case, the shell will then begin reading commands from the terminal, prompting with '% '. Processing of arguments and the use of the shell to process files containing command scripts will be described later.

The shell then repeatedly performs the following actions: a line of command input is read and broken into *words*. This sequence of words is placed on the command history list and then parsed. Finally each command in the current line is executed.

When a login shell terminates it executes commands from the file '.logout' in the users home directory.

Lexical structure

The shell splits input lines into words at blanks and tabs with the following exceptions. The characters '&' '|' ';' '<' '>' '(' ')' form separate words. If doubled in '&&', '||', '<<' or '>>' these pairs form single words. These parser metacharacters may be made part of other words, or prevented their special meaning, by preceding them with '\'. A newline preceded by a '\' is equivalent to a blank.

In addition strings enclosed in matched pairs of quotations, '`', '"' or '', form parts of a word; metacharacters in these strings, including blanks and tabs, do not form separate words. These quotations have semantics to be described subsequently. Within pairs of " or `` characters a newline preceded by a '\' gives a true newline character.

When the shell's input is not a terminal, the character '#' introduces a comment which continues to the end of the input line. It is prevented this special meaning when preceded by '\' and in quotations using '`', '"', and ''.

Commands

A simple command is a sequence of words, the first of which specifies the command to be executed. A simple command or a sequence of simple commands separated by '|' characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next. Sequences of pipelines may be separated by ';', and are then executed sequentially. A sequence of pipelines may be executed without waiting for it to terminate by following it with an '&'. Such a sequence is automatically prevented from being terminated by a hangup signal; the *nohup* command need not be used.

Any of the above may be placed in '(' ')' to form a simple command (which may be a component of a pipeline, etc.) It is also possible to separate pipelines with '||' or '&&' indicating, as in the C language, that the second is to be executed only if the first fails or succeeds respectively. (See *Expressions.*)

Substitutions

We now describe the various transformations the shell performs on the input in the order in which they occur.

History substitutions

History substitutions can be used to reintroduce sequences of words from previous commands, possibly performing modifications on these words. Thus history substitutions provide a generalization of a *redo* function.

History substitutions begin with the character '!' and may begin anywhere in the input stream if a history substitution is not already in progress. This '!' may be preceded by an '\' to prevent its special meaning; a '!' is passed unchanged when it is followed by a blank, tab, newline, '=' or '('. History substitutions also occur when an input line begins with '^'. This special abbreviation will be described later.

Any input line which contains history substitution is echoed on the terminal before it is executed as it could have been typed without history substitution.

Commands input from the terminal which consist of one or more words are saved on the history list, the size of which is controlled by the *history* variable. The previous command is always retained. Commands are numbered sequentially from 1.

For definiteness, consider the following output from the history command:

```
 9 write michael
10 ex write.c
11 cat oldwrite.c
12 diff *write.c
```

The commands are shown with their event numbers. It is not usually necessary to use event numbers, but the current event number can be made part of the *prompt* by placing an '!' in the prompt string.

With the current event 13 we can refer to previous events by event number '!11', relatively as in '!-2' (referring to the same event), by a prefix of a command word as in '!d' for event 12 or '!w' for event 9, or by a string contained in a word in the command as in '!?mic?' also referring to event 9. These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case '!!' refers to the previous command; thus '!!' alone is essentially a *redo*. The form '!#' references the current command (the one being typed in). It allows a word to be selected from further left in the line, to avoid retyping a long name, as in '!#:1'.

To select words from an event we can follow the event specification by a ':' and a designator for the desired words. The words of a input line are numbered from 0, the first (usually command) word being 0, the second word (first argument) being 1, etc. The basic word designators are:

0	first (command) word
n	*n*'th argument
^	first argument, i.e. '1'
$	last argument
%	word matched by (immediately preceding) ?*s*? search
x-*y*	range of words
-*y*	abbreviates '0-*y*'
*	abbreviates '^-$', or nothing if only 1 word in event
*x**	abbreviates '*x*-$'

x- like '*x*-' but omitting word '$'

The ':' separating the event specification from the word designator can be omitted if the argument selector begins with a '^', '$', '*' '-' or '%'. After the optional word designator can be placed a sequence of modifiers, each preceded by a ':'. The following modifiers are defined:

h	Remove a trailing pathname component, leaving the head.
r	Remove a trailing '.xxx' component, leaving the root name.
s/*l*/*r*/	Substitute *l* for *r*
t	Remove all leading pathname components, leaving the tail.
&	Repeat the previous substitution.
g	Apply the change globally, prefixing the above, e.g. 'g&'.
p	Print the new command but do not execute it.
q	Quote the substituted words, preventing further substitutions.
x	Like q, but break into words at blanks, tabs and newlines.

Unless preceded by a 'g' the modification is applied only to the first modifiable word. In any case it is an error for no word to be applicable.

The left hand side of substitutions are not regular expressions in the sense of the editors, but rather strings. Any character may be used as the delimiter in place of '/'; a '\' quotes the delimiter into the *l* and *r* strings. The character '&' in the right hand side is replaced by the text from the left. A '\' quotes '&' also. A null *l* uses the previous string either from a *l* or from a contextual scan string *s* in '!?*s*?'. The trailing delimiter in the substitution may be omitted if a newline follows immediately as may the trailing '?' in a contextual scan.

A history reference may be given without an event specification, e.g. '!$'. In this case the reference is to the previous command unless a previous history reference occurred on the same line in which case this form repeats the previous reference. Thus '!?foo?^ !$' gives the first and last arguments from the command matching '?foo?'.

A special abbreviation of a history reference occurs when the first non-blank character of an input line is a '^'. This is equivalent to '!:s^' providing a convenient shorthand for substitutions on the text of the previous line. Thus '^lb^lib' fixes the spelling of 'lib' in the previous command. Finally, a history substitution may be surrounded with '{' and '}' if necessary to insulate it from the characters which follow. Thus, after 'ls -ld ~paul' we might do '!{l}a' to do 'ls -ld ~paula', while '!la' would look for a command starting 'la'.

Quotations with ' and "

The quotation of strings by '"' and '`' can be used to prevent all or some of the remaining substitutions. Strings enclosed in '"' are prevented any further interpretation. Strings enclosed in '`' are yet variable and command expanded as described below.

In both cases the resulting text becomes (all or part of) a single word; only in one special case (see *Command Substitution* below) does a '`' quoted string yield parts of more than one word; '"' quoted strings never do.

Alias substitution

The shell maintains a list of aliases which can be established, displayed and modified by the *alias* and *unalias* commands. After a command line is scanned, it is parsed into distinct commands and the first word of each command, left-to-right, is checked to see if it has an alias. If it does, then the text which is the

alias for that command is reread with the history mechanism available as though that command were the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, then the argument list is left unchanged.

Thus if the alias for 'ls' is 'ls -l' the command 'ls /usr' would map to 'ls -l /usr', the argument list here being undisturbed. Similarly if the alias for 'lookup' was 'grep !^ /etc/passwd' then 'lookup bill' would map to 'grep bill /etc/passwd'.

If an alias is found, the word transformation of the input text is performed and the aliasing process begins again on the reformed input line. Looping is prevented if the first word of the new text is the same as the old by flagging it to prevent further aliasing. Other loops are detected and cause an error.

Note that the mechanism allows aliases to introduce parser metasyntax. Thus we can 'alias print 'pr \!* | lpr'' to make a command which *pr*'s its arguments to the line printer.

Variable substitution

The shell maintains a set of variables, each of which has as value a list of zero or more words. Some of these variables are set by the shell or referred to by it. For instance, the *argv* variable is an image of the shell's argument list, and words of this variable's value are referred to in special ways.

The values of variables may be displayed and changed by using the *set* and *unset* commands. Of the variables referred to by the shell a number are toggles; the shell does not care what their value is, only whether they are set or not. For instance, the *verbose* variable is a toggle which causes command input to be echoed. The setting of this variable results from the -v command line option.

Other operations treat variables numerically. The '@' command permits numeric calculations to be performed and the result assigned to a variable. Variable values are, however, always represented as (zero or more) strings. For the purposes of numeric operations, the null string is considered to be zero, and the second and subsequent words of multiword values are ignored.

After the input line is aliased and parsed, and before each command is executed, variable substitution is performed keyed by '$' characters. This expansion can be prevented by preceding the '$' with a '\' except within ""s where it always occurs, and within 's where it never occurs. Strings quoted by '` are interpreted later (see *Command substitution* below) so '$' substitution does not occur there until later, if at all. A '$' is passed unchanged if followed by a blank, tab, or end-of-line.

Input/output redirections are recognized before variable expansion, and are variable expanded separately. Otherwise, the command name and entire argument list are expanded together. It is thus possible for the first (command) word to this point to generate more than one word, the first of which becomes the command name, and the rest of which become arguments.

Unless enclosed in "" or given the ':q' modifier the results of variable substitution may eventually be command and filename substituted. Within "" a variable whose value consists of multiple words expands to a (portion of) a single word, with the words of the variables value separated by blanks. When the ':q' modifier is applied to a substitution the variable will expand to multiple words with each word separated by a blank and quoted to prevent later command or filename substitution.

The following metasequences are provided for introducing variable values into the shell input. Except as noted, it is an error to reference a variable which is not set.

$name
${name}
Are replaced by the words of the value of variable *name*, each separated by a blank. Braces insulate *name* from following characters which would otherwise be part of it. Shell variables have names consisting of up to 20 letters, digits, and underscores.

If *name* is not a shell variable, but is set in the environment, then that value is returned (but : modifiers and the other forms given below are not available in this case).

$name[selector]
${name[selector]}
May be used to select only some of the words from the value of *name*. The selector is subjected to '$' substitution and may consist of a single number or two numbers separated by a '-'. The first word of a variables value is numbered '1'. If the first number of a range is omitted it defaults to '1'. If the last member of a range is omitted it defaults to '$#name'. The selector '*' selects all words. It is not an error for a range to be empty if the second argument is omitted or in range.

$#name
${#name}
Gives the number of words in the variable. This is useful for later use in a '[selector]'.

$0
Substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

$number
${number}
Equivalent to '$argv[number]'.

$*
Equivalent to '$argv[*]'.

The modifiers ':h', ':t', ':r', ':q' and ':x' may be applied to the substitutions above as may ':gh', ':gt' and ':gr'. If braces '{' '}' appear in the command form then the modifiers must appear within the braces. The current implementation allows only one ':' modifier on each '$' expansion.

The following substitutions may not be modified with ':' modifiers.

$?name
${?name}
Substitutes the string '1' if name is set, '0' if it is not.

$?0
Substitutes '1' if the current input filename is know, '0' if it is not.

$$
Substitute the (decimal) process number of the (parent) shell.

Command and filename substitution

The remaining substitutions, command and filename substitution, are applied selectively to the arguments of builtin commands. This means that portions of expressions which are not evaluated are not subjected to these expansions. For commands which are not internal to the shell, the command name is substituted separately from the argument list. This occurs very late, after input-output redirection is performed, and in a child of the main shell.

Command substitution

Command substitution is indicated by a command enclosed in "`". The output from such a command is normally broken into separate words at blanks, tabs and newlines, with null words being discarded, this text then replacing the original string. Within "`"s, only newlines force new words; blanks and tabs are preserved.

In any case, the single final newline does not force a new word. Note that it is thus possible for a command substitution to yield only part of a word, even if the command outputs a complete line.

Filename substitution

If a word contains any of the characters '*', '?', '[' or '{' or begins with the character '~', then that word is a candidate for filename substitution, also known as 'globbing'. This word is then regarded as a pattern, and replaced with an alphabetically sorted list of file names which match the pattern. In a list of words specifying filename substitution it is an error for no pattern to match an existing file name, but it is not required for each pattern to match. Only the metacharacters '*', '?' and '[' imply pattern matching, the characters '~' and '{' being more akin to abbreviations.

In matching filenames, the character '.' at the beginning of a filename or immediately following a '/', as well as the character '/' must be matched explicitly. The character '*' matches any string of characters, including the null string. The character '?' matches any single character. The sequence '[...]' matches any one of the characters enclosed. Within '[...]', a pair of characters separated by '-' matches any character lexically between the two.

The character '~' at the beginning of a filename is used to refer to home directories. Standing alone, i.e. '~' it expands to the invokers home directory as reflected in the value of the variable *home*. When followed by a name consisting of letters, digits and '-' characters the shell searches for a user with that name and substitutes their home directory; thus '~ken' might expand to '/usr/ken' and '~ken/chmach' to '/usr/ken/chmach'. If the character '~' is followed by a character other than a letter or '/' or appears not at the beginning of a word, it is left undisturbed.

The metanotation 'a{b,c,d}e' is a shorthand for 'abe ace ade'. Left to right order is preserved, with results of matches being sorted separately at a low level to preserve this order. This construct may be nested. Thus '~source/s1/{oldls,ls}.c' expands to '/usr/source/s1/oldls.c /usr/source/s1/ls.c' whether or not these files exist without any chance of error if the home directory for 'source' is '/usr/source'. Similarly '../{memo,*box}' might expand to '../memo ../box ../mbox'. (Note that 'memo' was not sorted with the results of matching '*box'.) As a special case '{', '}' and '{}' are passed undisturbed.

Input/output

The standard input and standard output of a command may be redirected with the following syntax:

< name

Open file *name* (which is first variable, command and filename expanded) as the standard input.

<< word

Read the shell input up to a line which is identical to *word*. *Word* is not subjected to variable, filename or command substitution, and each input line is compared to *word* before any substitutions are done on this input line. Unless a quoting '\', '"', '' or '`' appears in *word* variable and command substitution is performed on the intervening lines, allowing '\' to quote '$', '\' and '`'. Commands which are substituted have all blanks, tabs, and newlines preserved, except for the final newline which is dropped. The resultant text is placed in an anonymous temporary file which is given to the command as standard input.

> name
>! name
>& name
>&! name

The file *name* is used as standard output. If the file does not exist then it is created; if the file exists, its is truncated, its previous contents being lost.

If the variable *noclobber* is set, then the file must not exist or be a character special file (e.g. a terminal or '/dev/null') or an error results. This helps prevent accidental destruction of files. In this case the '!' forms can be used and suppress this check.

The forms involving '&' route the diagnostic output into the specified file as well as the standard output. *Name* is expanded in the same way as '<' input filenames are.

>> name
>>& name
>>! name
>>&! name

Uses file *name* as standard output like '>' but places output at the end of the file. If the variable *noclobber* is set, then it is an error for the file not to exist unless one of the '!' forms is given. Otherwise similar to '>'.

If a command is run detached (followed by '&') then the default standard input for the command is the empty file '/dev/null'. Otherwise the command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Thus, unlike some previous shells, commands run from a file of shell commands have no access to the text of the commands by default; rather they receive the original standard input of the shell. The '<<' mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block read its input.

Diagnostic output may be directed through a pipe with the standard output. Simply use the form '|&' rather than just '|'.

Expressions

A number of the builtin commands (to be described subsequently) take expressions, in which the operators are similar to those of C, with the same precedence. These expressions appear in the *@*, *exit*, *if*, and *while* commands. The following operators are available:

| || && | ^ & == != <= >= < > << >> + - * / % ! ~ ( )

Here the precedence increases to the right, '==' and '!=', '<=' '>=' '<' and '>', '<<' and '>>', '+' and '-', '*' '/' and '%' being, in groups, at the same level. The '==' and '!=' operators compare their arguments as strings, all others operate on numbers. Strings which begin with '0' are considered octal numbers. Null or missing arguments are considered '0'. The result of all expressions are strings, which represent decimal numbers. It is important to note that no two components of an expression can appear in the same word; except when adjacent to components of expressions which are syntactically significant to the parser ('&' '|' '<' '>' '(' ')') they should be surrounded by spaces.

Also available in expressions as primitive operands are command executions enclosed in '{' and '}' and file enquiries of the form '-*l* name' where *l* is one of:

r	read access
w	write access
x	execute access
e	existence
o	ownership
z	zero size
f	plain file
d	directory

The specified name is command and filename expanded and then tested to see if it has the specified relationship to the real user. If the file does not exist or is inaccessible then all enquiries return false, i.e. '0'. Command executions succeed, returning true, i.e. '1', if the command exits with status 0, otherwise they fail, returning false, i.e. '0'. If more detailed status information is required then the command should be executed outside of an expression and the variable *status* examined.

Control flow

The shell contains a number of commands which can be used to regulate the flow of control in command files (shell scripts) and (in limited but useful ways) from terminal input. These commands all operate by forcing the shell to reread or skip in its input and, due to the implementation, restrict the placement of some of the commands.

The *foreach*, *switch*, and *while* statements, as well as the *if-then-else* form of the *if* statement require that the major keywords appear in a single simple command on an input line as shown below.

If the shell's input is not seekable, the shell buffers up input whenever a loop is being read and performs seeks in this internal buffer to accomplish the rereading implied by the loop. (To the extent that this allows, backward goto's will succeed on non-seekable inputs.)

Builtin commands

Builtin commands are executed within the shell. If a builtin command occurs as any component of a pipeline except the last then it is executed in a subshell.

alias
alias name
alias name wordlist
The first form prints all aliases. The second form prints the alias for name. The final form assigns the specified *wordlist* as the alias of *name*; *wordlist* is command and filename substituted. *Name* is not allowed to be *alias* or *unalias*

alloc
Shows the amount of dynamic core in use, broken down into used and free core, and address of the last location in the heap. With an argument shows each used and free block on the internal dynamic memory chain indicating its address, size, and whether it is used or free. This is a debugging command and may not work in production versions of the shell; it requires a modified version of the system memory allocator.

break
Causes execution to resume after the *end* of the nearest enclosing *foreall* or *while*. The remaining commands on the current line are executed. Multi-level breaks are thus possible by writing them all on one line.

breaksw
Causes a break from a *switch*, resuming after the *endsw*.

case label:
A label in a *switch* statement as discussed below.

cd
cd name
chdir
chdir name
Change the shells working directory to directory *name*. If no argument is given then change to the home directory of the user.

If *name* is not found as a subdirectory of the current directory (and does not begin with '/', './', or '../'), then each component of the variable *cdpath* is checked to see if it has a subdirectory *name*. Finally, if all else fails but *name* is a shell variable whose value begins with '/', then this is tried to see if it is a directory.

continue
Continue execution of the nearest enclosing *while* or *foreach*. The rest of the commands on the current line are executed.

default:
Labels the default case in a *switch* statement. The default should come after all *case* labels.

echo wordlist
The specified words are written to the shells standard output. A '\c' causes the echo to complete without printing a newline, akin to the '\c' in *nroff*(1). A '\n' in wordlist causes a newline to be printed. Otherwise the words are echoed, separated by spaces.

else
end
endif
endsw
See the description of the *foreach*, *if*, *switch*, and *while* statements below.

exec command
The specified command is executed in place of the current shell.

exit
exit(expr)
The shell exits either with the value of the *status* variable (first form) or with the value of the specified *expr* (second form).

foreach name (wordlist)
...
end
The variable *name* is successively set to each member of *wordlist* and the sequence of commands between this command and the matching *end* are executed. (Both *foreach* and *end* must appear alone on separate lines.)

The builtin command *continue* may be used to continue the loop prematurely and the builtin command *break* to terminate it prematurely. When this command is read from the terminal, the loop is read up once prompting with '?' before any statements in the loop are executed. If you make a mistake typing in a loop at the terminal you can rub it out.

glob wordlist
Like *echo* but no '\' escapes are recognized and words are delimited by null characters in the output. Useful for programs which wish to use the shell to filename expand a list of words.

goto word
The specified *word* is filename and command expanded to yield a string of the form 'label'. The shell rewinds its input as much as possible and searches for a line of the form 'label:' possibly preceded by blanks or tabs. Execution continues after the specified line.

history
Displays the history event list.

if (expr) command
If the specified expression evaluates true, then the single *command* with arguments is executed. Variable substitution on *command* happens early, at the same time it does for the rest of the *if* command. *Command* must be a simple command, not a pipeline, a command list, or a parenthesized command list. Input/output redirection occurs even if *expr* is false, when command is not executed (this is a bug).

if (expr) **then**
...
else if (expr2) **then**
...
else
...
endif
If the specified *expr* is true then the commands to the first *else* are executed; else if *expr2* is true then the commands to the second else are

executed, etc. Any number of *else-if* pairs are possible; only one *endif* is needed. The *else* part is likewise optional. (The words *else* and *endif* must appear at the beginning of input lines; the *if* must appear alone on its input line or after an *else*.)

login
Terminate a login shell, replacing it with an instance of **/bin/login**. This is one way to log off, included for compatibility with **/bin/sh**.

logout
Terminate a login shell. Especially useful if *ignoreeof* is set.

nice
nice +number
nice command
nice +number command
The first form sets the *nice* for this shell to 4. The second form sets the *nice* to the given number. The final two forms run command at priority 4 and *number* respectively. The super-user may specify negative niceness by using 'nice -number ...'. Command is always executed in a sub-shell, and the restrictions place on commands in simple *if* statements apply.

nohup
nohup command
The first form can be used in shell scripts to cause hangups to be ignored for the remainder of the script. The second form causes the specified command to be run with hangups ignored. On the Computer Center systems at UC Berkeley, this also *submits* the process. Unless the shell is running detached, *nohup* has no effect. All processes detached with "&" are automatically *nohup'ed*. (Thus, *nohup* is not really needed.)

onintr
onintr -
onintr label
Control the action of the shell on interrupts. The first form restores the default action of the shell on interrupts which is to terminate shell scripts or to return to the terminal command input level. The second form 'onintr -' causes all interrupts to be ignored. The final form causes the shell to execute a 'goto label' when an interrupt is received or a child process terminates because it was interrupted.

In any case, if the shell is running detached and interrupts are being ignored, all forms of *onintr* have no meaning and interrupts continue to be ignored by the shell and all invoked commands.

rehash
Causes the internal hash table of the contents of the directories in the *path* variable to be recomputed. This is needed if new commands are added to directories in the *path* while you are logged in. This should only be necessary if you add commands to one of your own directories, or if a systems programmer changes the contents of one of the system directories.

repeat count command
The specified *command* which is subject to the same restrictions as the *command* in the one line *if* statement above, is executed *count* times. I/O redirections occurs exactly once, even if *count* is 0.

set
set name
set name=word
set name[index]=word
set name=(wordlist)
The first form of the command shows the value of all shell variables. Variables which have other than a single word as value print as a parenthesized word list. The second form sets *name* to the null string. The third form sets *name* to the single *word*. The fourth form sets the *index*'th component of name to word; this component must already exist. The final form sets *name* to the list of words in *wordlist*. In all cases the value is command and filename expanded.

These arguments may be repeated to set multiple values in a single set command. Note however, that variable expansion happens for all arguments before any setting occurs.

setenv name value
(Version 7 systems only.) Sets the value of environment variable *name* to be *value*, a single string. Useful environment variables are 'TERM' the type of your terminal and 'SHELL' the shell you are using.

shift
shift variable
The members of *argv* are shifted to the left, discarding *argv*[*1*]. It is an error for *argv* not to be set or to have less than one word as value. The second form performs the same function on the specified variable.

source name
The shell reads commands from *name*. *Source* commands may be nested; if they are nested too deeply the shell may run out of file descriptors. An error in a *source* at any level terminates all nested *source* commands. Input during *source* commands is never placed on the history list.

switch (string)
case str1:
...
breaksw
...
default:
...
breaksw
endsw
Each case label is successively matched, against the specified *string* which is first command and filename expanded. The file metacharacters '*', '?' and '[...]' may be used in the case labels, which are variable expanded. If none of the labels match before a 'default' label is found, then the execution begins after the default label. Each case label and the default label must appear at the beginning of a line. The command *breaksw* causes execution to continue after the *endsw*. Otherwise control may fall through case labels and default labels as in C. If no label matches and there is no default, execution continues after the *endsw*.

time

time command
With no argument, a summary of time used by this shell and its children is printed. If arguments are given the specified simple command is timed and a time summary as described under the *time* variable is printed. If necessary, an extra shell is created to print the time statistic when the command completes.

umask
umask value
The file creation mask is displayed (first form) or set to the specified value (second form). The mask is given in octal. Common values for the mask are 002 giving all access to the group and read and execute access to others or 022 giving all access except no write access for users in the group or others.

unalias pattern
All aliases whose names match the specified pattern are discarded. Thus all aliases are removed by 'unalias *'. It is not an error for nothing to be *unaliased*.

unhash
Use of the internal hash table to speed location of executed programs is disabled.

unset pattern
All variables whose names match the specified pattern are removed. Thus all variables are removed by 'unset *'; this has noticeably distasteful side-effects. It is not an error for nothing to be *unset*.

wait
All child processes are waited for. If the shell is interactive, then an interrupt can disrupt the wait, at which time the shell prints names and process numbers of all children known to be outstanding.

while (expr)
...
end
While the specified expression evaluates non-zero, the commands between the *while* and the matching *end* are evaluated. *Break* and *continue* may be used to terminate or continue the loop prematurely. (The *while* and *end* must appear alone on their input lines.) Prompting occurs here the first time through the loop as for the *foreach* statement if the input is a terminal.

@
@ name = expr
@ name[index] = expr
The first form prints the values of all the shell variables. The second form sets the specified *name* to the value of *expr*. If the expression contains '<', '>', '&' or '|' then at least this part of the expression must be placed within '(' ')'. The third form assigns the value of *expr* to the *index*'th argument of *name*. Both *name* and its *index*'th component must already exist.

The operators '*=', '+=', etc are available as in C. The space separating the name from the assignment operator is optional. Spaces are, however, mandatory in separating components of *expr* which would otherwise be single words.

Special postfix '++' and '--' operators increment and decrement *name* respectively, i.e. '@ i++'.

Pre-defined variables

The following variables have special meaning to the shell. Of these, *argv*, *child*, *home*, *path*, *prompt*, *shell* and *status* are always set by the shell. Except for *child* and *status* this setting occurs only at initialization; these variables will not then be modified unless this is done explicitly by the user.

The shell copies the environment variable PATH into the variable *path*, and copies the value back into the environment whenever *path* is set. Thus is is not necessary to worry about its setting other than in the file *.cshrc* as inferior *csh* processes will import the definition of *path* from the environment. (It could be set once in the *.login* except that commands through *net*(1) would not see the definition.)

argv Set to the arguments to the shell, it is from this variable that positional parameters are substituted, i.e. '$1' is replaced by '$argv[1]', etc.

cdpath Gives a list of alternate directories searched to find subdirectories in *chdir* commands.

child The process number printed when the last command was forked with '&'. This variable is *unset* when this process terminates.

echo Set when the -x command line option is given. Causes each command and its arguments to be echoed just before it is executed. For non-builtin commands all expansions occur before echoing. Builtin commands are echoed before command and filename substitution, since these substitutions are then done selectively.

histchars Can be assigned a two character string. The first character is used as a history character in place of "!", the second character is used in place of the "^" substitution mechanism. For example, "set histchars=",;"" will cause the history characters to be comma and semicolon.

history Can be given a numeric value to control the size of the history list. Any command which has been referenced in this many events will not be discarded. Too large values of *history* may run the shell out of memory. The last executed command is always saved on the history list.

home The home directory of the invoker, initialized from the environment. The filename expansion of '~' refers to this variable.

ignoreeof If set the shell ignores end-of-file from input devices which are terminals. This prevents shells from accidentally being killed by control-D's.

mail The files where the shell checks for mail. This is done after each command completion which will result in a prompt, if a specified interval has elapsed. The shell says 'You have new mail.' if the file exists with an access time not greater than its modify time.

If the first word of the value of *mail* is numeric it specifies a different mail checking interval, in seconds, than the default, which is 10 minutes.

If multiple mail files are specified, then the shell says 'New mail in *name*' when there is mail in the file *name*.

noclobber As described in the section on *Input/output*, restrictions are placed on output redirection to insure that files are not accidentally destroyed, and that '>>' redirections refer to existing files.

noglob If set, filename expansion is inhibited. This is most useful in shell scripts which are not dealing with filenames, or after a list of filenames has been obtained and further expansions are not desirable.

nonomatch If set, it is not an error for a filename expansion to not match any existing files; rather the primitive pattern is returned. It is still an error for the primitive pattern to be malformed, i.e. 'echo [' still gives an error.

path Each word of the path variable specifies a directory in which commands are to be sought for execution. A null word specifies the current directory. If there is no *path* variable then only full path names will execute. The usual search path is '.', '/bin' and '/usr/bin', but this may vary from system to system. For the super-user the default search path is '/etc', '/bin' and '/usr/bin'. A shell which is given neither the -c nor the -t option will normally hash the contents of the directories in the *path* variable after reading *.cshrc*, and each time the *path* variable is reset. If new commands are added to these directories while the shell is active, it may be necessary to give the *rehash* or the commands may not be found.

prompt The string which is printed before each command is read from an interactive terminal input. If a '!' appears in the string it will be replaced by the current event number unless a preceding '\' is given. Default is '% ', or '# ' for the super-user.

shell The file in which the shell resides. This is used in forking shells to interpret files which have execute bits set, but which are not executable by the system. (See the description of *Non-builtin Command Execution* below.) Initialized to the (system-dependent) home of the shell.

status The status returned by the last command. If it terminated abnormally, then 0200 is added to the status. Builtin commands which fail return exit status '1', all other builtin commands set status '0'.

time Controls automatic timing of commands. If set, then any command which takes more than this many cpu seconds will cause a line giving user, system, and real times and a utilization percentage which is the ratio of user plus system times to real time to be printed when it terminates.

verbose Set by the -v command line option, causes the words of each command to be printed after history substitution.

Non-builtin command execution

When a command to be executed is found to not be a builtin command the shell attempts to execute the command via *exec*(2). Each word in the variable *path* names a directory from which the shell will attempt to execute the command. If

it is given neither a -c nor a -t option, the shell will hash the names in these directories into an internal table so that it will only try an *exec* in a directory if there is a possibility that the command resides there. This greatly speeds command location when a large number of directories are present in the search path. If this mechanism has been turned off (via *unhash*), or if the shell was given a -c or -t argument, and in any case for each directory component of *path* which does not begin with a "/", the shell concatenates with the given command name to form a path name of a file which it then attempts to execute.

Parenthesized commands are always executed in a subshell. Thus '(cd ; pwd) ; pwd' prints the *home* directory; leaving you where you were (printing this after the home directory), while 'cd ; pwd' leaves you in the *home* directory. Parenthesized commands are most often used to prevent *chdir* from affecting the current shell.

If the file has execute permissions but is not an executable binary to the system, then it is assumed to be a file containing shell commands an a new shell is spawned to read it.

If there is an *alias* for *shell* then the words of the alias will be prepended to the argument list to form the shell command. The first word of the *alias* should be the full path name of the shell (e.g. '$shell'). Note that this is a special, late occurring, case of *alias* substitution, and only allows words to be prepended to the argument list without modification.

Argument list processing

If argument 0 to the shell is '-' then this is a login shell. The flag arguments are interpreted as follows:

- -c Commands are read from the (single) following argument which must be present. Any remaining arguments are placed in *argv*.
- -e The shell exits if any invoked command terminates abnormally or yields a non-zero exit status.
- -f The shell will start faster, because it will neither search for nor execute commands from the file '.cshrc' in the invokers home directory.
- -i The shell is interactive and prompts for its top-level input, even if it appears to not be a terminal. Shells are interactive without this option if their inputs and outputs are terminals.
- -n Commands are parsed, but not executed. This may aid in syntactic checking of shell scripts.
- -s Command input is taken from the standard input.
- -t A single line of input is read and executed. A '\' may be used to escape the newline at the end of this line and continue onto another line.
- -v Causes the *verbose* variable to be set, with the effect that command input is echoed after history substitution.
- -x Causes the *echo* variable to be set, so that commands are echoed immediately before execution.
- -V Causes the *verbose* variable to be set even before '.cshrc' is executed.
- -X Is to -x as -V is to -v.

After processing of flag arguments if arguments remain but none of the -c, -i, -s, or -t options was given the first argument is taken as the name of a file of commands to be executed. The shell opens this file, and saves its name for possible resubstitution by '$0'. Since many systems use either the standard version 6 or version 7 shells whose shell scripts are not compatible with this shell, the shell will execute such a 'standard' shell if the first character of a script is not a '#', i.e. if the script does not start with a comment. Remaining arguments initialize the variable *argv*.

Signal handling

The shell normally ignores *quit* signals. The *interrupt* and *quit* signals are ignored for an invoked command if the command is followed by '&'; otherwise the signals have the values which the shell inherited from its parent. The shells handling of interrupts can be controlled by *onintr*. Login shells catch the *terminate* signal; otherwise this signal is passed on to children from the state in the shell's parent. In no case are interrupts allowed when a login shell is reading the file '.logout'.

AUTHOR

William Joy

FILES

~/.cshrc	Read at beginning of execution by each shell.
~/.login	Read by login shell, after '.cshrc' at login.
~/.logout	Read by login shell, at logout.
/bin/sh	Standard shell, for shell scripts not starting with a '#'.
/tmp/sh*	Temporary file for '<<'.
/dev/null	Source of empty file.
/etc/passwd	Source of home directories for '~name'.

LIMITATIONS

Words can be no longer than 512 characters. The number of characters in an argument varies from system to system. Early version 6 systems typically have 512 character limits while later version 6 and version 7 systems have 5120 character limits. The number of arguments to a command which involves filename expansion is limited to 1/6'th the number of characters allowed in an argument list. Also command substitutions may substitute no more characters than are allowed in an argument list.

To detect looping, the shell restricts the number of *alias* substitutions on a single line to 20.

SEE ALSO

access(2), exec(2), fork(2), pipe(2), signal(2), umask(2), wait(2), a.out(5), environ(5), 'An introduction to the C shell'

BUGS

Control structure should be parsed rather than being recognized as built-in commands. This would allow control commands to be placed anywhere, to be combined with '|', and to be used with '&' and ';' metasyntax.

Commands within loops, prompted for by '?', are not placed in the *history* list.

It should be possible to use the ':' modifiers on the output of command substitutions. All and more than one ':' modifier should be allowed on '$' substitutions.

Some commands should not touch *status* or it may be so transient as to be almost useless. Oring in 0200 to *status* on abnormal termination is a kludge.

In order to be able to recover from failing *exec* commands on version 6 systems, the new command inherits several open files other than the normal standard input and output and diagnostic output. If the input and output are redirected and the new command does not close these files, some files may be held open unnecessarily.

There are a number of bugs associated with the importing/exporting of the PATH. For example, directories in the path using the ~ syntax are not expanded in the PATH. Unusual paths, such as (), can cause csh to core dump.

This version of *csh* does not support or use the process control features of the 4th Berkeley Distribution. It contains a number of known bugs which have been fixed in the process control version. This version is not supported.

NAME

digest -- create menu system(s) for the Business Shell

SYNOPSIS

digest [options] menufile ...

DESCRIPTION

Digest is used to create a menu system for use by the Business Shell (bsh(1)). This program is also used to modify an existing menu system.

One or more menu systems may be created under control of the options described below:

-h Display an informative summary of the available options and defaults. -a is the same as -h.

-l number
Check for menus longer than number lines in length. The default value is 25 if none is specified. This is the correct maximum number for a conventional 24-line crt screen. In general, number should be one larger than the length of the screen area (as defined by "li" in termcap) for the terminal to be used. The user is responsible for ensuring that the width of a menu will fit onto the terminal(s) he uses. Bsh(1) will truncate lines which are too wide (without issuing a warning message).

-m Multiple menu systems: For each menu file (which must be a directory), produce a separate menu system. The names for each menu system are created by suffixing ".bin" to the menu file name.

-s menu
The starting menu for the generated menu system is the one specified. This option doesn't make much sense if used with the -m option. If no starting menu is specified, the alphabetically first menu name is used for each menu system.

-v Verbose: echo menu names as they are processed.

-o file
The digested output is sent to the named file. By convention, a digested menu system file name should end with a ".bin" suffix.

A menu file may contain one or more menus or directories containing menus. Digest will recursively process all menus within a directory structure.

Note that the -m and -o options are mutually exclusive. The -m option indicates that each menu is to produce a separate ".bin" file: -o indicates that a single output file is to be produced with the name given.

The default output file is "menul.bin" if none is specified via the -o option, where "menul" is the first menu file name.

The recommended way to create a menu system is to create a tree of directories containing the various portions of the system. Each subtree contains all the menus related to a given subject. Thus, a primary menu (directory) is created for, say, system management functions and subsidiary menus are placed beneath (within) the directory for each of the individual system management functions or function areas. Help menus may be placed wherever appropriate in the structure.

SEE ALSO

bsh(1), menus(5), termcap(5)

DIAGNOSTICS

The diagnostics produced by digest are intended to be self-explanatory.

BUGS

No outstanding bugs are known.

Digest might check each menu for validity and each menu system for consistency.

NAME

edit – text editor (variant of the ex editor for new or casual users)

SYNOPSIS

edit [**–r**] name ...

DESCRIPTION

Edit is a variant of the text editor *ex* recommended for new or casual users who wish to use a command oriented editor. The following brief introduction should help you get started with *edit*. A more complete basic introduction is provided by *Edit: A tutorial* . A *Ex/edit command summary (version 2.0)* is also very useful. See *ex*(UCB) for other useful documents; in particular, if you are using a CRT terminal you will want to learn about the display editor *vi*.

BRIEF INTRODUCTION

To edit the contents of an existing file you begin with the command "edit name" to the shell. *Edit* makes a copy of the file which you can then edit, and tells you how many lines and characters are in the file. To create a new file, just make up a name for the file and try to run *edit* on it; you will cause an error diagnostic, but don't worry.

Edit prompts for commands with the character ":", which you should see after starting the editor. If you are editing an existing file, then you will have some lines in *edit*'s buffer (its name for the copy of the file you are editing). Most commands to *edit* use its "current line" if you don't tell them which line to use. Thus if you say print (which can be abbreviated p) and hit carriage return (as you should after all *edit* commands) this current line will be printed. If you delete (d) the current line, *edit* will print the new current line. When you start editing, *edit* makes the last line of the file the current line. If you delete this last line, then the new last line becomes the current one. In general, after a delete, the next line in the file becomes the current line. (Deleting the last line is a special case.)

If you start with an empty file, or wish to add some new lines, then the append (a) command can be used. After you give this command (typing a carriage return after the word append) *edit* will read lines from your terminal until you give a line consisting of just a ".", placing these lines after the current line. The last line you type then becomes the current line. The command insert (i) is like append but places the lines you give before, rather than after, the current line.

Edit numbers the lines in the buffer, with the first line having number 1. If you give the command "1" then *edit* will type this first line. If you then give the command delete *edit* will delete the first line, and line 2 will become line 1, and *edit* will print the current line (the new line 1) so you can see where you are. In general, the current line will always be the last line affected by a command.

You can make a change to some text within the current line by using the substitute (s) command. You say "s/*old*/*new*/" where *old* is replaced by the old characters you want to get rid of and *new* is the new characters you want to replace it with.

The command file (f) will tell you how many lines there are in the buffer you are editing and will say "[Modified]" if you have changed it. After modifying a file you can put the buffer text back to replace the file by giving a write (w) command. You can then leave the editor by issuing a quit (q) command. If you run *edit* on a file, but don't change it, it is not necessary (but does no harm) to write the file back. If you try to quit from *edit* after modifying the buffer without

writing it out, you will be warned that there has been "No write since last change" and *edit* will await another command. If you wish not to write the buffer out then you can issue another quit command. The buffer is then irretrievably discarded, and you return to the shell.

By using the delete and append commands, and giving line numbers to see lines in the file you can make any changes you desire. You should learn at least a few more things, however, if you are to use *edit* more than a few times.

The change (c) command will change the current line to a sequence of lines you supply (as in append you give lines up to a line consisting of only a "."). You can tell change to change more than one line by giving the line numbers of the lines you want to change, i.e. "3,5change". You can print lines this way too. Thus "1,23p" prints the first 23 lines of the file.

The undo (u) command will reverse the effect of the last command you gave which changed the buffer. Thus if give a substitute command which doesn't do what you want, you can say undo and the old contents of the line will be restored. You can also undo an undo command so that you can continue to change your mind. *Edit* will give you a warning message when commands you do affect more than one line of the buffer. If the amount of change seems unreasonable, you should consider doing an *undo* and looking to see what happened. If you decide that the change is ok, then you can *undo* again to get it back. Note that commands such as *write* and *quit* cannot be undone.

To look at the next line in the buffer you can just hit carriage return. To look at a number of lines hit ^D (control key and, while it is held down D key, then let up both) rather than carriage return. This will show you a half screen of lines on a CRT or 12 lines on a hardcopy terminal. You can look at the text around where you are by giving the command "z.". The current line will then be the last line printed; you can get back to the line where you were before the "z." command by saying "''". The z command can also be given other following characters "z-" prints a screen of text (or 24 lines) ending where you are; "z+" prints the next screenful. If you want less than a screenful of lines do, e.g., "z.12" to get 12 lines total. This method of giving counts works in general; thus you can delete 5 lines starting with the current line with the command "delete 5".

To find things in the file you can use line numbers if you happen to know them; since the line numbers change when you insert and delete lines this is somewhat unreliable. You can search backwards and forwards in the file for strings by giving commands of the form /*text*/ to search forward for *text* or ?*text*? to search backward for *text*. If a search reaches the end of the file without finding the text it wraps, end around, and continues to search back to the line where you are. A useful feature here is a search of the form /^*text*/ which searches for *text* at the beginning of a line. Similarly /*text*$/ searches for *text* at the end of a line. You can leave off the trailing / or ? in these commands.

The current line has a symbolic name "."; this is most useful in a range of lines as in "..$print" which prints the rest of the lines in the file. To get to the last line in the file you can refer to it by its symbolic name "$". Thus the command "$ delete" or "$d" deletes the last line in the file, no matter which line was the current line before. Arithmetic with line references is also possible. Thus the line "$-5" is the fifth before the last, and ".+20" is 20 lines after the present.

You can find out which line you are at by doing ".=". This is useful if you wish to move or copy a section of text within a file or between files. Find out the first and last line numbers you wish to copy or move (say 10 to 20). For a move you

can then say "10,20move "a" which deletes these lines from the file and places them in a buffer named a. *Edit* has 26 such buffers named a through z. You can later get these lines back by doing ""a move ." to put the contents of buffer a after the current line. If you want to move or copy these lines between files you can give an edit (e) command after copying the lines, following it with the name of the other file you wish to edit, i.e. "edit chapter2". By changing *move* to *copy* above you can get a pattern for copying lines. If the text you wish to move or copy is all within one file then you can just say "10,20move $" for example. It is not necessary to use named buffers in this case (but you can if you wish).

SEE ALSO

ex (UCB), vi (UCB), 'Edit: A tutorial', by Ricki Blau and James Joyce

AUTHOR

William Joy

BUGS

See *ex*(UCB).

NAME

ex – text editor

SYNOPSIS

ex [**–**] [**–v**] [**–t** tag] [**–r**] [*+lineno*] name ...

DESCRIPTION

Ex is the root of a family of editors: *edit*, *ex* and *vi*. *Ex* is a superset of *ed*, with the most notable extension being a display editing facility. Display based editing is the focus of *vi*.

If you have not used *ed*, or are a casual user, you will find that the editor *edit* is convenient for you. It avoids some of the complexities of *ex* used mostly by systems programmers and persons very familiar with *ed*.

If you have a CRT terminal, you may wish to use a display based editor; in this case see *vi*(UCB), which is a command which focuses on the display editing portion of *ex*.

DOCUMENTATION

For *edit* and *ex* see the *Ex/edit command summary – Version 2.0*. The document *Edit: A tutorial* provides a comprehensive introduction to *edit* assuming no previous knowledge of computers or the UNIX system.

The *Ex Reference Manual – Version 2.0* is a comprehensive and complete manual for the command mode features of *ex*, but you cannot learn to use the editor by reading it. For an introduction to more advanced forms of editing using the command mode of *ex* see the editing documents written by Brian Kernighan for the editor *ed*; the material in the introductory and advanced documents works also with *ex*.

An Introduction to Display Editing with Vi introduces the display editor *vi* and provides reference material on *vi*. The *Vi Quick Reference* card summarizes the commands of *vi* in a useful, functional way, and is useful with the *Introduction*.

FOR ED USERS

If you have used *ed* you will find that *ex* has a number of new features useful on CRT terminals. Intelligent terminals and high speed terminals are very pleasant to use with *vi*. Generally, the editor uses far more of the capabilities of terminals than *ed* does, and uses the terminal capability data base *termcap*(UCB) and the type of the terminal you are using from the variable TERM in the environment to determine how to drive your terminal efficiently. The editor makes use of features such as insert and delete character and line in its visual command (which can be abbreviated vi) and which is the central mode of editing when using *vi*(UCB). There is also an interline editing open (o) command which works on all terminals.

Ex contains a number of new features for easily viewing the text of the file. The **z** command gives easy access to windows of text. Hitting ^D causes the editor to scroll a half-window of text and is more useful for quickly stepping through a file than just hitting return. Of course, the screen oriented visual mode gives constant access to editing context.

Ex gives you more help when you make mistakes. The **undo** (**u**) command allows you to reverse any single change which goes astray. *Ex* gives you a lot of feedback, normally printing changed lines, and indicates when more than a few lines are affected by a command so that it is easy to detect when a command has affected more lines than it should have.

The editor also normally prevents overwriting existing files unless you edited them so that you don't accidentally clobber with a *write* a file other than the one you are editing. If the system (or editor) crashes, or you accidentally hang up the phone, you can use the editor recover command to retrieve your work. This will get you back to within a few lines of where you left off.

Ex has several features for dealing with more than one file at a time. You can give it a list of files on the command line and use the **next** (n) command to deal with each in turn. The **next** command can also be given a list of file names, or a pattern as used by the shell to specify a new set of files to be dealt with. In general, filenames in the editor may be formed with full shell metasyntax. The metacharacter '%' is also available in forming filenames and is replaced by the name of the current file. For editing large groups of related files you can use *ex*'s **tag** command to quickly locate functions and other important points in any of the files. This is useful when working on a large program when you want to quickly find the definition of a particular function. The command *ctags*(UCB) builds a *tags* file or a group of C programs.

For moving text between files and within a file the editor has a group of buffers, named a through z. You can place text in these named buffers and carry it over when you edit another file.

There is a command **&** in *ex* which repeats the last **substitute** command. In addition there is a confirmed substitute command. You give a range of substitutions to be done and the editor interactively asks whether each substitution is desired.

You can use the **substitute** command in *ex* to systematically convert the case of letters between upper and lower case. It is possible to ignore case of letters in searches and substitutions. *Ex* also allows regular expressions which match words to be constructed. This is convenient, for example, in searching for the word "edit" if your document also contains the word "editor."

Ex has a set of *options* which you can set to tailor it to your liking. One option which is very useful is the *autoindent* option which allows the editor to automatically supply leading white space to align text. You can then use the ^D key as a backtab and space and tab forward to align new code easily.

Miscellaneous new useful features include an intelligent **join** (j) command which supplies white space between joined lines automatically, commands < and > which shift groups of lines, and the ability to filter portions of the buffer through commands such as *sort*.

FILES

/usr/lib/ex2.0strings	error messages
/usr/lib/ex2.0recover	recover command
/usr/lib/ex2.0preserve	preserve command
/etc/termcap	describes capabilities of terminals
~/.exrc	editor startup file
/tmp/Exnnnnnn	editor temporary
/tmp/Rxnnnnnn	named buffer temporary
/usr/preserve	preservation directory

SEE ALSO

awk(1), ed(1), grep(1), sed(1), edit(UCB), grep(UCB), termcap(UCB), vi(UCB)

AUTHOR

William Joy

BUGS

The *undo* command causes all marks to be lost on lines changed and then restored if the marked lines were changed.

Undo never clears the buffer modified condition.

The z command prints a number of logical rather than physical lines. More than a screen full of output may result if long lines are present.

File input/output errors don't print a name if the command line '-' option is used.

There is no easy way to do a single scan ignoring case.

Because of the implementation of the arguments to *next*, only 512 bytes of argument list are allowed there.

The format of */etc/termcap* and the large number of capabilities of terminals used by the editor cause terminal type setup to be rather slow.

The editor does not warn if text is placed in named buffers and not used before exiting the editor.

Null characters are discarded in input files, and cannot appear in resultant files.

To calculate the number of test cases required for branch coverage, you need to cover all the edges (lines).

So from the above flow chart, we can cover all the lines / edges with two test cases.

For getting the correct answer browse down to the end of this page

<<<<<< ========== >>>>>>

). 48: **Use the following code;**

=1
:2
3
x == 2)

nt "Hi There ";

NAME

format - format a floppy disk while running XENIX

SYNOPSIS

format

DESCRIPTION

Format is a menu-driven program for formatting floppy disks. Diskettes are formatted in Altos 5-1/4 inch format; double-density, double-sided.

The options to layout are used to create some very common layouts.

USAGE

```
layout /dev/hd0.layout 586
```

SEE ALSO

map(1), sizefs(1)

-t If fsck cannot obtain enough memory to keep its tables, it uses a scratch file. If the -t is specified, the file named in the next argument is used as the scratch file. Without the -t option, fsck prompts if it needs a scratch file. The file should not be on the file system being checked, and if it is not a special file or did not already exist, it is removed when fsck completes.

If no filesystems are given to fsck then a default list of file systems is read from the file /etc/checklist.

Inconsistencies checked are as follows:

1. Blocks claimed by more than one i-node or the free list.

2. Blocks claimed by an i-node or the free list outside the range of the file system.

3. Incorrect link counts.

4. Size checks:

   Incorrect number of blocks in file.
   Directory size not a multiple of 16 bytes.

5. Bad i-node format.

6. Blocks not accounted for anywhere.

7. Directory checks:

   File pointing to unallocated i-node.
   I-node number out of range.

8. Super Block checks:

   More than 65536 i-nodes.
   More blocks for i-nodes than there are in the file system.

9. Bad free block list format.

10. Total free block and/or free i-node count incorrect.

Orphaned files and directories (allocated but unreferenced) are, with the operator's concurrence, reconnected by placing them in the "lost+found" directory. The name assigned is the i-node number. The only restriction is that the directory "lost+found" must preexist in the root of the filesystem being checked and must have empty slots in which entries can be made. This is accomplished by making "lost+found", copying a number of files to the directory, and then removing them (before fsck is executed).

-u Use time of last access instead of last modification for sorting (-t) or printing (-l).

-c Use time of file creation for sorting or printing.

-i Print i-number in first column of the report for each file listed.

-f Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off -l, -t, -s, and -r, and turns on -a; the order is the order in which entries appear in the directory.

-g Give group ID instead of owner ID in long listing.

-m Force stream output format.

-l Force one entry per line output format, e.g. to a teletype.

-c Force multi-column output, e.g. to a file or a pipe.

-q Force printing of non-graphic characters in file names as the character '?'; this normally happens only if the output device is a teletype.

-b Force printing of non-graphic characters to be in the `ddd notation in octal.

-x Force columnar printing to be sorted across rather than down the page; this is the default if the last character of the name the program is invoked with is an 'x'.

-f Cause directories to be marked with a trailing '/' and executable files to be marked with a trailing '*'; this is the default if the last character of the name the program is invoked with is a 'f'.

-R Recursively list subdirectories encountered.

The mode printed under the -l option contains 11 characters which are interpreted as follows: the first character is

d if the entry is a directory;
b if the entry is a block-type special file;
c if the entry is a character-type special file;
m if the entry is a multiplexor-type character special file;
- if the entry is a plain file.

NAME

layout - configure a hard disk

SYNOPSIS

layout layout-device 586

DESCRIPTION

Layout creates a table defining a number of "logical devices" associated with each physical disk in the XENIX system. Layout records this table on cylinder zero of each disk. Each entry in the table is in the following format:

```
struct layout {
     daddr_t l_blkoff; /*Block offset to area */
     daddr_t l_nblocks; /*Number of blocks in area */
};
```

Layout defines ten "logical devices" on the hard disk:

0 The whole disk, with the alternate sector mechanism disabled.

1 The swap area.

2 The root file system.

3-8 Unused.

9 Alternate sector area into which bad disk sectors are automatically mapped by the XENIX kernel.

The logical device numbers above correspond to device numbers in the hard disk driver.

Other device numbers are pre-defined in the XENIX kernel as follows:

10 Future expansion.

11 All of track0.

12 Boot program area.

13 Portion of cylinder zero used for fsck temporary file.

14 Layout information created by this utility.

15 Alternate sector map (see map(1)).

NAME

map - create an alternate sector map for a hard disk drive

SYNOPSIS

map layout mapfile drive

DESCRIPTION

Map creates a bad sector map, on mapfile, using the layout information, in layout, created by layout(1). The last argument is the logical device name which references the whole drive.

The standard invocation is:

```
map /dev/hd0.layout  /dev/hd0.secmap  /dev/hd0
```

The structure used for the bad sector to alternate sector mapping is as follows:

```
struct mapsec {
     int   bad_cyl;  /* Cylinder number of bad sector */
     char  bad_hed;  /* Head number of bad sector */
     char  bad_sec;  /* Sector number of bad sector */
     int   bad_good; /* Offset into alternate sector
                        area */
};
```

This structure provides a way for the XENIX hard disk driver to recover from bad sectors it encounters when reading the hard disk. If a bad sector is read, a search of a table of the above structures is made. If an exact match of cylinder, head and sector is found, the corresponding offset is used as an index into the area reserved on the disk for alternate sectors.

SEE ALSO

layout(1), sizefs(1)

NAME

ls - List contents of directory

SYNOPSIS

ls {-abcdfgilmqrstuxLCFR} name...

DESCRIPTION

For each directory argument, ls lists the contents of the directory; for each file argument, ls repeats its name and any other information requested. The output is sorted alphabetically by defalt. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments appear before directories and their contents.

There are three major listing formats. The format chosen depends on whether the output is going to a teletype, and may also be controlled by option flags. The default format for a teletype is to list the contents of directories in multi-column format, with the entries sorted down the columns. (Files which are not the contents of a directory being interpreted are always sorted across the page rather than down the page in columns. This is because the individual file names may be arbitrarily long.) If the standard output is not a teletype, the default format is to list one entry per line. Finally, there is a stream output format in which files are listed across the page, separated by ',' characters. The -m flag enables this format; when invoked as l this format is also used.

There are an unbelievable number of options:

-l List in long format, giving mode, number of links, owner, size in bytes, and time of last modification for each file. (See below.) If the file is a special file the size field will instead contain the major and minor device numbers.

-t Sort by time modified (latest first) instead of by name, as is normal.

-a List all entries; usually '.' and '..' are suppressed.

-s Give size in blocks, including indirect blocks, for each entry.

-d If argument is a directory, list only its name, not its contents (mostly used with -l to get status on directory).

-r Reverse the order of sort to get reverse alphabetic or oldest first as appropriate.

Personal and systemwide distribution lists. It is also possible to create a personal distribution lists so that, for instance, you can send mail to "cohorts" and have it go to a group of people. Such lists can be defined by placing a line like

alias cohorts bill ozalp sklower jkf mark cory:kridle

in the file .mailrc in your home directory. The current list of such aliases can be displayed by the alias (a) command in *mail*. System wide distribution lists can be created by editing /usr/lib/aliases, see *aliases*(5) and *delivermail*(8); these are kept in a slightly different syntax. In mail you send, personal aliases will be expanded in mail sent to others so that they will be able to reply to the recipients. System wide *aliases* are not expanded when the mail is sent, but any reply returned to the machine will have the system wide alias expanded as all mail goes through *delivermail*. If you edit /usr/lib/aliases, you must run the program *newaliases*(1).

Network mail (ARPA, UUCP, Berknet) Mail to sites on the ARPA network and sites within Bell laboratories can be sent using "name@site" for ARPA-net sites or "machine!user" for Bell labs sites, provided appropriate gateways are known to the system. (Be sure to escape the ! in Bell sites when giving it on a *csh* command line by preceding it with an \. Machines on an instance of the Berkeley network are addressed as "machine:user", e.g. "csvax:bill". When addressed from the arpa-net, "csvax:bill" is known as "csvax.bill@berkeley".

Mail has a number of options which can be set in the *.mailrc* file to alter its behavior; thus "set askcc" enables the "askcc" feature. (These options are summarized below.)

SUMMARY

(Adapted from the 'Mail Reference Manual') Each command is typed on a line by itself, and may take arguments following the command word. The command need not be typed in its entirety – the first command which matches the typed prefix is used. For the commands which take message lists as arguments, if no message list is given, then the next message forward which satisfies the command's requirements is used. If there are no messages forward of the current message, the search proceeds backwards, and if there are no good messages at all, *mail* types "No applicable messages" and aborts the command.

– Goes to the previous message and prints it out. If given a numeric argument *n*, goes to the *n th* previous message and prints it.

? Prints a brief summary of commands.

! Executes the UNIX shell command which follows.

alias (a) With no arguments, prints out all currently-defined aliases. With one argument, prints out that alias. With more than one argument, adds the users named in the second and later arguments to the alias named in the first argument.

chdir (c) Changes the user's working directory to that specified, if given. If no directory is given, then changes to the user's login directory.

delete (d) Takes a list of messages as argument and marks them all as deleted. Deleted messages will not be saved in *mbox*, nor will they be available for most other commands.

dp (also dt) Deletes the current message and prints the next message. If there is no next message, *mail* says "at EOF."

edit (e) Takes a list of messages and points the text editor at each one in

The next 9 characters are interpreted as three sets of three bits each. The first set refers to owner permissions; the next to permissions to others in the same user-group; and the last to all others. Within each set the three characters indicate permission respectively to read, to write, or to execute the file as a program. For a directory, 'execute' permission is interpreted to mean permission to search the directory for a specified file. The permissions are indicated as follows:

r if the file is readable;
w if the file is writable;
x if the file is executable;
- if the indicated permission is not granted.

The group-execute permission character is given as s if the file has set-group-ID mode; likewise the user-execute permission character is given as s if the file has set-user-ID mode.

The last character of the mode (normally 'x' or '-') is t if the 1000 bit of the mode is on. See chmod(1) for the meaning of this mode.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks is printed.

FILES

/etc/passwd to get user ID's for 'ls-l'
/etc/group to get group ID's for 'ls-g'

BUGS

Newline and tab are considered printing characters in file names.

The output device is assumed to be 80 columns wide.

The option setting based on whether the output is a teletype is undesirable as "ls-s" is much different than "ls-s|lpr". On the other hand, not doing this setting would make old shell scripts which used ls almost certain losers.

significance.

undelete (u) Takes a message list and marks each one as *not* being deleted.

unset Takes a list of option names and discards their remembered values; the inverse of **set**.

visual (v) Takes a message list and invokes the display editor on each message.

write (w) A synonym for **save**.

xit (x) A synonym for **exit**.

Here is a summary of the tilde escapes, which are used when composing messages to perform special functions. Tilde escapes are only recognized at the beginning of lines. The name "tilde escape" is somewhat of a misnomer since the actual escape character can be set by the option **escape**.

~!command
Execute the indicated shell command, then return to the message.

~c name ...
Add the given names to the list of carbon copy recipients.

~d Read the file "dead.letter" from your home directory into the message.

~e Invoke the text editor on the message collected so far. After the editing session is finished, you may continue appending text to the message.

~h Edit the message header fields by typing each one in turn and allowing the user to append text to the end or modify the field by using the current terminal erase and kill characters.

~m messages
Read the named messages into the message being sent, shifted right one tab. If no messages are specified, read the current message.

~p Print out the message collected so far, prefaced by the message header fields.

~q Abort the message being sent, copying the message to "dead.letter" in your home directory if **save** is set.

~r filename
Read the named file into the message.

~s string Cause the named string to become the current subject field.

~t name ...
Add the given names to the direct recipient list.

~v Invoke an alternate editor (defined by the VISUAL option) on the message collected so far. Usually, the alternate editor will be a screen editor. After you quit the editor, you may resume appending text to the end of your message.

~w filename
Write the message onto the named file.

~|command
Pipe the message through the command as a filter. If the command gives no output or terminates abnormally, retain the original text of

NAME

mail – send and receive mail

SYNOPSIS

mail [**–f** [name]] [people ...]

INTRODUCTION

Mail is a intelligent mail processing system, which has a command syntax reminiscent of *ed* with lines replaced by messages.

Sending mail. To send a message to one or more other people, *mail* can be invoked with arguments which are the names of people to send to. You are then expected to type in your message, followed by an EOT (control–D) at the beginning of a line. The section below, labeled *Replying to or originating mail*, describes some features of *mail* available to help you compose your letter.

Reading mail. In normal usage, *mail* is given no arguments and checks your mail out of the post office, then printing out a one line header of each message there. The current message is initially the first message (numbered 1) and can be printed using the **print** command (which can be abbreviated **p**). You can move among the messages much as you move between lines in *ed*, with the commands '+' and '–' moving backwards and forwards, and simple numbers typing the addressed message.

Disposing of mail. After examining a message you can **delete** (**d**) the message or **reply** (**r**) to it. Deletion causes the *mail* program to forget about the message. This is not irreversible, the message can be **undelete**d (**u**) by giving its number, or the *mail* session can be aborted by giving the **exit** (**x**) command. Deleted messages will, however, usually disappear never to be seen again.

Specifying messages. Commands such as **print** and **delete** often can be given a list of message numbers as argument to apply to a number of messages at once. Thus "delete 1 2" deletes messages 1 and 2, while "delete 1–5" deletes messages 1 through 5. The special name "*" addresses all messages, and "$" addresses the last message; thus the command **top** which prints the first few lines of a message could be used in "top *" to print the first few lines of all messages.

Replying to or originating mail. You can use the **reply** command to set up a response to a message, sending it back to the person who it was from. Text you then type in, up to an end-of-file (or a line consisting only of a ".") defines the contents of the message. While you are composing a message, *mail* treats lines beginning with the character '~' specially. For instance, typing "~m" (alone on a line) will place a copy of the current message into the response right shifting it by a tabstop. Other escapes will set up subject fields, add and delete recipients to the message and allow you to escape to an editor to revise the message or to a shell to run some commands. (These options will be given in the summary below.)

Ending a mail processing session. You can end a *mail* session with the **quit** (**q**) command. Messages which have been examined go to your *mbox* file unless they have been deleted in which case they are discarded. Unexamined messages go back to the post office. The **–f** option causes *mail* to read in the contents of your *mbox* (or the specified file) for processing; when you quit *mail* writes undeleted messages back to this file.

/tmp/R#	temporary for editor escape
/usr/lib/Mail.help*	help files
/usr/lib/Mail.rc	system initialization file
/bin/mail	to do actual mailing
/etc/delivermail	postman

SEE ALSO

biamail(1), fmt(1), newaliases(1), aliases(5), delivermail(8)
'The Mail Reference Manual'

AUTHOR

Kurt Shoens

BUGS

turn. On return from the editor, the message is read back in.

exit (**ex** or **x**) Effects an immediate return to the Shell without modifying the user's system mailbox, his *mbox* file, or his edit file in **-f** .

from (**f**) Takes a list of messages and prints their message headers.

headers (**h**) Lists the current range of headers, which is an 18 message group. If a "+" argument is given, then the next 18 message group is printed, and if a "-" argument is given, the previous 18 message group is printed.

help A synonym for ?

hold (**ho**, also **preserve**) Takes a message list and marks each message therein to be saved in the user's system mailbox instead of in *mbox*. Does not override the **delete** command.

mail (**m**) Takes as argument login names and distribution group names and sends mail to those people.

next (**n** like + or CR) Goes to the next message in sequence and types it. With an argument list, types the next matching message.

preserve A synonym for **hold**.

print (**p**) Takes a message list and types out each message on the user's terminal.

quit (**q**) Terminates the session, saving all undeleted, unsaved messages in the user's *mbox* file in his login directory, preserving all messages marked with **hold** or **preserve** or never referenced in his system mailbox, and removing all other messages from his system mailbox. If new mail has arrived during the session, the message "You have new mail" is given. If given while editing a mailbox file with the **-f** flag, then the edit file is rewritten. A return to the Shell is effected, unless the rewrite of edit file fails, in which case the user can escape with the **exit** command.

reply (**r**) Takes a message list and sends mail to each message author just like the **mail** command. The default message must not be deleted.

respond A synonym for **reply** .

save (**s**) Takes a message list and a filename and appends each message in turn to the end of the file. The filename in quotes, followed by the line count and character count is echoed on the user's terminal.

set (**se**) With no arguments, prints all variable values. Otherwise, sets option. Arguments are of the form "option=value" or "option."

shell (**sh**) Invokes an interactive version of the shell.

size Takes a message list and prints out the size in characters of each message.

top Takes a message list and prints the top few lines of each. The number of lines printed is controlled by the variable **toplines** and defaults to five.

type (**t**) A synonym for **print** .

unalias Takes a list of names defined by **alias** commands and discards the remembered groups of users. The group names no longer have any

NAME

ps – process status

SYNOPSIS

ps [acgklrstuvwx# [corefile] [swapfile] [system]]

DESCRIPTION

Ps prints certain indicia about active processes. To get a complete printout on the console or lpr, use "ps axlgw" For a quick snapshot of system activity, "ps au" is recommended. A minus may precede options with no effect. The following options may be specified.

a asks for information about all processes with terminals (ordinarily only one's own processes are displayed).

c causes only the *comm* field to be displayed instead of the arguments. (The comm field is the tail of the path name of the file the process last exec'ed.) This option speeds up *ps* somewhat and reduces the amount of output. It is also more reliable since the process can't scribble on top of it.

g Asks for all processes. Without this option, *ps* only prints "interesting" processes. Processes are deemed to be uninteresting if they are process group leaders, or if their arguments begin with a '–'. This normally eliminates shells and getty processes.

k causes the file */usr/sys/core* is used in place of */dev/kmem* and */dev/mem*. This is used for postmortem system debugging.

l asks for a long listing. The short listing contains the user name, process ID, tty, the cumulative execution time of the process and an approximation to the command line.

r asks for "raw output". A non-human readable sequence of structures is output on the standard output. There is one structure for each process, the format is defined by <psout.h>

s Print the size of the kernel stack of each process. This may only be used with the short listing, and is for use by system developers.

t*ttyname*

restricts output to processes whose controlling tty is the specified ttyname (which should be specified as printed by *ps*, e.g. *tty3* for tty3, *tconsole* for console, *ttyd0* for ttyd0, *t?* for processes with no tty, etc). This option must be the last one given.

u A user oriented output is produced. This includes the name of the owner of the process, process id, nice value, size, resident set size, tty, cpu time used, and the command.

w tells *ps* you are on a wide terminal (132 columns). *Ps* normally assumes you are on an 80 column terminal. This information is used to decide how much of long commands to print. The w option may be repeated, e.g. ww, and the entire command, up to 128 characters, will be printed without regard to terminal width.

x asks even about processes with no terminal.

A process number may be given, (indicated here by *#*), in which case the output is restricted to that process. This option must also be last.

the message. The command *fmt*(1) is often used as *command* to rejustify the message.

~~*string* Insert the *string* of text in the message prefaced by a single ~. If you have changed the escape character, then you should double that character in order to send it.

Options are controlled via the **set** and **unset** commands. Options may be either binary, in which case it is only significant to see whether they are set or not, or string, in which case the actual value is of interest. The binary options include the following:

append Causes messages saved in *mbox* to be appended to the end rather than prepended. (This is set in /usr/lib/Mail.rc on version 7 systems.)

ask 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.

askcc Causes you to be prompted for additional carbon copy recipients at the end of each message. Responding with a newline indicates your satisfaction with the current list.

autoprint Causes the **delete** command to behave like **dp** – thus, after deleting a message, the next one will be typed automatically.

ignore Causes interrupt signals from your terminal to be ignored and echoed as @'s.

metoo Usually, when a group is expanded that contains the sender, the sender is removed from the expansion. Setting this option causes the sender to be included in the group.

quiet Suppresses the printing of the version when first invoked.

save Causes the message collected prior to a interrupt to be saved on the file "dead.letter" in your home directory on receipt of two interrupts (or after a ~q.)

The following options have string values:

EDITOR Pathname of the text editor to use in the **edit** command and ~e escape. If not defined, then a default editor is used.

SHELL Pathname of the shell to use in the ! command and the ~! escape. A default shell is used if this option is not defined.

VISUAL Pathname of the text editor to use in the **visual** command and ~v escape.

escape If defined, the first character of this option gives the character to use in the place of ~ to denote escapes.

record If defined, gives the pathname of the file used to record all outgoing mail. If not defined, then outgoing mail is not so saved.

toplines If defined, gives the number of lines of a message to be printed out with the **top** command; normally, the first five lines are printed.

FILES

/usr/spool/mail/*	post office
~/mbox	your old mail
~/.mailrc	file giving initial mail commands

Processes with large environments, which have all or part of the command in a block other than the top block in memory, are not correctly printed by ps, which only looks at the top block in memory. Thus, users using the TERMCAP environment variable will probably only have their command name shown.

NAME

printenv – print out the environment

SYNOPSIS

printenv [name]

DESCRIPTION

Printenv prints out the values of the variables in the environment. If a *name* is specified, only its value is printed.

If a *name* is specified and it is not defined in the environment, *printenv* returns exit status 1, else it returns status 0.

SEE ALSO

sh(1), environ(5), csh(UCB)

BUGS

A second argument tells *ps* where to look for *core* if the *k* option is given, instead of /vmcore. A third argument is the name of a swap file to use instead of the default /dev/drum. If a fourth argument is given, it is taken to be the file containing the system's namelist. Otherwise, "/vmunix" is used.

The output is sorted by tty, then by process ID.

The long listing is columnar and contains

F Flags associated with the process. These are defined by #define lines in /usr/include/sys/proc.h.

S The state of the process. 0: nonexistent; S: sleeping; W: waiting; R: running; I: intermediate; Z: terminated; T: stopped.

UID The user id of the process owner.

PID The process ID of the process; as in certain cults it is possible to kill a process if you know its true name.

PPID The process ID of the parent process.

CPU Processor utilization for scheduling.

PRI The priority of the process; high numbers mean low priority.

NICE Used in priority computation.

ADDR The memory address of the process if resident, otherwise the disk address.

SZ The size in blocks of the memory image of the process.

WCHAN
The event for which the process is waiting or sleeping; if blank, the process is running.

TTY The controlling tty for the process.

TIME The cumulative execution time for the process.

COMMAND
The command and its arguments.

A process that has exited and has a parent, but has not yet been waited for by the parent is marked <defunct>. *Ps* makes an educated guess as to the file name and arguments given when the process was created by examining memory or the swap area. The method is inherently somewhat unreliable and in any event a process is entitled to destroy this information, so the names cannot be counted on too much.

FILES

/vmunix	system namelist
/dev/kmem	kernel memory
/dev/drum	swap device
/vmcore	core file
/dev	searched to find swap device and tty names

SEE ALSO

kill(1), w(1)

BUGS

Things can change while *ps* is running; the picture it gives is only a close approximation to reality.

NAME

multiuser - bring the system up multiuser

SYNOPSIS

multiuser

DESCRIPTION

Multiuser prompts the user to set the current system date and time, and then brings the system up multiuser.

First, multiuser displays the current system date and time and asks the user to confirm or change the date and then the time. Confirmation is done by entering Return. The format for entering the date is "yymmdd." Time is entered as a 24-hour clock in the form "hhmm."

SEE ALSO

date(1)

NAME

reset – reset the teletype bits to a sensible state

SYNOPSIS

reset

DESCRIPTION

Reset sets the teletype bits to 'soft-copy terminal standard mode' with the erase character set to control-h and the kill character to '@'. *Reset* is most useful when you crap out in raw mode.

SEE ALSO

stty(1), stty(2), gtty(2)

AUTHOR

Kurt Shoens

BUGS

If you are in a funny state you may well have to type "reset" followed by linefeed (control-j if there is no such key.)

NAME

tar - tape or floppy archiver

SYNOPSIS

tar [key] [name ...]

DESCRIPTION

Tar saves and restores files on magtape or floppy. Its actions are controlled by the key argument. The key is a string of characters containing at most one function letter and possibly one or more function modifiers. Other arguments to the command are file or directory names specifying which files are to be dumped or restored. In all cases, appearance of a directory name refers to the files and (recursively) subdirectories of that directory.

Note that XENIX contains a new version of tar, which permits a file to extend across media boundaries. For compatability considerations with the previous version of tar, refer to the BUGS section below.

The function portion of the key is specified by one of the following letters:

r The named files are written on the end of the tape. The c function implies this.

x The named files are extracted from the tape. If the named file matches a directory whose contents had been written onto the tape, this directory is (recursively) extracted. The owner and mode are restored (if possible). If no file argument is given, the entire content of the tape or floppy is extracted. Note that if multiple entries specifying the same file are on the tape, the last version will overwrite all preceeding versions.

t The names of the specified files are listed each time they occur on the tape. If no file argument is given, all of the names on the tape are listed.

u The named files are added to the tape if either they are not already there or have been modified since last put on the tape.

c Create a new tape; writing begins on the beginning of the tape instead of after the last file. This command implies r.

The following characters may be used in addition to the letter which selects the function desired.

NAME

map - create an alternate sector map for a hard disk drive

SYNOPSIS

map layout mapfile drive

DESCRIPTION

Map creates a bad sector map, on mapfile, using the layout information, in layout, created by layout(1). The last argument is the logical device name which references the whole drive.

The standard invocation is:

```
map /dev/hd0.layout  /dev/hd0.secmap  /dev/hd0
```

The structure used for the bad sector to alternate sector mapping is as follows:

```
struct mapsec {
     int   bad_cyl;  /* Cylinder number of bad sector */
     char  bad_hed;  /* Head number of bad sector */
     char  bad_sec;  /* Sector number of bad sector */
     int   bad_good; /* Offset into alternate sector
                        area */
};
```

This structure provides a way for the XENIX hard disk driver to recover from bad sectors it encounters when reading the hard disk. If a bad sector is read, a search of a table of the above structures is made. If an exact match of cylinder, head and sector is found, the corresponding offset is used as an index into the area reserved on the disk for alternate sectors.

SEE ALSO

layout(1), sizefs(1)

The b option should not be used with archives that are going to be updated. If the archive is on a disk file, the b option should not be used at all, as updating an archive stored in this manner can destroy it.
The current limit on file name length is 100 characters.

EXAMPLES

To dump the directory /usr/john to diskette(s), enter the command

```
tar cvf /dev/fd0/usr/john
```

Note that if the device /dev/tar has been configured to reference the floppy disk drive, as desired, the above command can be abbreviated to:

```
tar cv /usr/john
```

NAME

sizefs - determine the size of a logical device from the layout information associated with a hard disk.

SYNOPSIS

sizefs layout-file logical-device-number

DESCRIPTION

Sizefs prints on the standard output the size in blocks of the specified area on the disk. It gets its information out of the structure created by layout(1). Its most common use is in shell scripts for creating a file system on the hard disk, where its output is used as an argument to mkfs(1).

SEE ALSO

layout(1), map(1), mkfs(1)

exists for a new user, it is not removed. All files under /etc/newuser are copied to the new directory during the user installation process. Typically /etc/newuser will contain the standard versions of the following files: .cshrc, .login, .logout, .profile. The initial value given to a new user ID is one more than the maximum user ID currently in use. The same is true for a new group ID.

Delete allows the deletion of an existing user or group. Deleting a user optionally also deletes his directory and all files contained within it. Deleting a user will not cause all files throughout the system owned by the user to be deleted -- only those beneath his directory. Thus, some files may have an "unknown" owner after a user is deleted. And, if a user is later added with the same user ID as the deleted user, these files will suddenly belong to the new user. The same problem may arise with the deletion and later addition of a group.

Show will show an individual user or group or all users or groups. The word "show" may be omitted if desired.

Change allows the modification of any existing user or group. A special display mode is entered with a menu of fields for selection of the item to be modified. Typing RETURN or LINE FEED in response to a field change request will empty the field. Changes to a user or group change the corresponding entries in the /etc/passwd and /etc/group files. Changing a user's directory entry will not cause a renaming of the actual directory. It is the user's responsibility to ensure that the /etc/passwd and /etc/group files remain consistent.

Help displays a short informative text on the screen. "?" is equivalent to help. The message is the same one as obtained by invoking ua with the "-h" option.

! escapes to the shell (see sh(1)). If no arguments are given, a shell is invoked which will continue until it receives an end-of-file. Then ua resumes. If arguments are present, a shell is invoked with the "-c" option and the arguments are passed along. Ua resumes immediately thereafter. If csh(1) is desired rather than sh(1), the command !csh should be used.

Quit immediately terminates ua and returns to the system.

Any command which is not understood by ua causes an appropriate message to be displayed. As a side-effect, the working portion of the screen is cleared.

Ua does not distinguish between RETURN and LINE FEED. They may be used interchangably.

If the screen becomes "dirty" for some reason, you can force

v Normally tar does its work silently. The v (verbose) option causes it to type the name of each file it treats preceded by the function letter. With the t function, v gives more information about the tape entries than just the name.

w Causes tar to print the action to be taken followed by file name, then wait for user confirmation. If a word beginning with 'y' is given, the action is performed. Any other input means don't do it.

f Causes tar to use the next argument as the name of the archive instead of /dev/tar. If the name of the file is '-', tar writes to standard output or reads from standard input, whichever is appropriate. Thus, tar can also be used to move hierarchies with the command

```
cd fromdir; tar cf - . | (cd todir; tar xf -)
```

b Causes tar to use the next argument as the blocking factor for tape records. The default is 1, the maximum is 8. This option should only be used with raw magnetic tape archives (see f above). Altos recommends a blocking factor of 8 when using the cartridge tape.

l Tells tar to complain if it cannot resolve all the links to the files dumped. If this is not specified, no error messages are printed.

s Obsolete. No longer supported. (Was size parameter, used when files did not cross diskette boundaries.)

FILES

/dev/tar default input/output device
/tmp/tar*

DIAGNOSTICS

Complains about bad key characters and tape read/write errors.
Complains if not enough memory is available to hold the link tables.
Tar will tell you to change volumes when the current volume (floppy or tape) becomes full. It expects you to type one or more characters and then return.

BUGS

This version of tar can read old style tar disks, and the old tar program can read new style tar disks, as long as they do not extend over multiple floppies.
Note that the old version of tar cannot be used to read multiple volume archives created by the new version of tar.
There is no way to ask for the n-th occurrence of a file.
Tape errors are handled ungracefully.
The u option can be slow.

DIAGNOSTICS

The diagnostics produced by ua are intended to be self-explanatory.

BUGS

Ua should allow specification of alternate passwd and group files.

Complete consistency checking between the /etc/passwd and /etc/group files is not done. In particular, it is possible to install a user with an unknown group in the passwd file and it is possible to install a group with an unknown user in the group file. The shells and directories specified in the /etc/passwd file are not checked for existence or accessibility.

Ua does not check for duplicated user IDs or duplicated group IDs.

Impossible user IDs and group IDs are permitted.

Impossible or non-existent names may be specified for a user's Directory and Shell fields.

The System 3 commands pwck(1M) and grpck(1m) should be incorporated into ua.

NOTE:

DO NOT USE ua TO SET A USER'S PASSWORD. The password would be incorrectly encrypted, and the user would NOT be able to log in successfully. Passwords may only be set with the passwd command, explained in PASSWD(1). The password field displayed by ua is the encrypted version contained in /etc/passwd.

NAME

ua -- user administration

SYNOPSIS

ua [-h]

DESCRIPTION

Ua is used for the addition, deletion and modification of users and groups. It provides an effective means for maintaining the system password (/etc/passwd) and system group (/etc/group) files.

The command is implemented using the termcap and curses facilities from UC Berkeley. It must be run interactively from a terminal which is defined within /etc/termcap.

This command should only be run by Super User -- improper results may occur if it is run by a normal user.

The following option is interpreted by ua:

-h Displays the program's current version and copyright notice as well as a short description of the program's functions.

Ua displays its legal commands at the top of the screen. The "Command?" prompt at the bottom of the screen indicates that ua is awaiting input. The command language syntax is:

```
[ add|delete|show|change ] [ user <name> | group <name> ]

show [ Users | Groups ]

[ help | ? ]

! [ <shell command(s)> ]

quit
```

The system responds as soon as the first letter of a command is typed. Full command words are not acceptable as input. The case of each word is significant: "group" is not the same as "Group."

Typing an interrupt (usually RUBOUT or DEL) will cause ua to immediately return to the top-level command interpreter.

Add allows the addition of a new user or group. After user/group is specified and a new name provided, the system immediately enters the change command so as to allow modification of the new entry. At the conclusion of the change command the addition is made. If a directory already

ua to clear it and redisplay the current contents by transmitting an ASCII "DC2." This is Control-r on most of the currently popular terminals.

Ua understands the Backspace function (as obtained from /etc/termcap). In addition, any time a word is partially formed, the ESCape key will cause the partial word to be discarded and input restarted.

Ua interprets the CANcel key to mean "terminate the current operation." The CANcel key is Control-x on many of the more popular terminals. The CANcel key is more powerful than ESCape, but not so powerful as "interrupt."

Ua will immediately return to the top-level command interpreter upon receipt of an interrupt signal. Such a signal is usually generated via the DEL, RUBOUT or BREAK key.

Ua creates a special user named "standard" in /etc/passwd if one is not already present. This entry is used as the template for installing new users. Thus, if it is desired to have all new users defaulted to the standard UNIX shell (/bin/sh) for the Shell field, it is only necessary to update the Shell field in the "standard" user.

Before adding a new user with a new group, the new group should be added. Otherwise, ua has no way to properly create the new entry in /etc/passwd since it contains group numbers rather than group names.

During program initialization ua copies /etc/passwd and /etc/group to /etc/opasswd and /etc/ogroup, respectively. Thus, if a mistake or disaster occurs during the use of this program, the user may recover the prior state of either or both files.

FILES

/etc/passwd	used for login name to user ID conversions
/etc/group	used for group name to group ID conversions
/etc/opasswd	this file is a copy of /etc/passwd before any modifications are made
/etc/ogroup	this file is a copy of /etc/group before any modifications are made
/etc/newuser	directory containing files which will be installed in a new user's account
/etc/termcap	contains terminal attribute descriptions
/tmp/passwd	temporary file
/tmp/group	temporary file
/etc/ua.lock	lock file

SEE ALSO

group(5), passwd(5)

FILES

/usr/include/user.h contains definitions for EACCESS and EDEADLOCK.

/usr/include/sys/locking.h contains definitions for UNLOCK, LOCK, NBLOCK, RLOCK, NBRLOCK.

NAME

vi – screen oriented (visual) display editor based on ex

SYNOPSIS

vi [–t tag] [–r] [+lineno] name ...

DESCRIPTION

vi (visual) is a display oriented text editor based on ex(UCB). *Ex* and *vi* run the same code; it is possible to get to the command mode of *ex* from within *vi* and vice-versa.

The *Vi Quick Reference* card and the *Introduction to Display Editing with Vi* provide full details on using *vi*.

FILES

See *ex*(UCB).

SEE ALSO

ex (UCB), vi (UCB), "Vi Quick Reference" card, "An Introduction to Display Editing with Vi".

BUGS

Scans with / and ? begin on the next line, skipping the remainder of the current line.

Software tabs using ^T work only immediately after the *autoindent*.

Left and right shifts on intelligent terminals don't make use of insert and delete character operations in the terminal.

The *wrapmargin* option can be fooled since it looks at output columns when blanks are typed. If a long word passes through the margin and onto the next line without a break, then the line won't be broken.

Insert/delete within a line can be slow if tabs are present on intelligent terminals, since the terminals need help in doing this correctly.

Occasionally inverse video scrolls up into the file from a diagnostic on the last line.

Saving text on deletes in the named buffers is somewhat inefficient.

The *source* command does not work when executed as :source; there is no way to use the :append, :change, and :insert commands, since it is not possible to give more than one line of input to a : escape. To use these on a :global you must Q to ex command mode, execute them, and then reenter the screen editor with *vi* or *open*.

NAME

locking - lock or unlock a record of a file

SYNOPSIS

locking(fildes, ltype, nbytes); int fildes, ltype, nbytes;

DESCRIPTION

locking performs a locking action **ltype** on a record of the open file specified by **fildes.** The record starts at the current file position and has a length of **nbytes.** If the value of nbytes is Ø, the entire file is locked. Nbytes may extend beyond the end of the file, in which case only the process issuing the lock call may access or add information to the file within the boundary defined by nbytes. Thus, lock defines a range in the file controlled by the locking process, and this control may extend to records that have yet to be added to the end of the file. The available values for ltype are:

UNLOCK	Ø	Unlock the record.
LOCK	1	Lock the given record; the calling process will sleep if any part of the record has been locked by a different process.
NBLOCK	2	Lock the given record; if any part of the record is already locked by a different process, return the error EACCESS.
RLOCK	3	Same as LOCK except that reading is allowed by other processes.
NBRLOCK	4	Same as NBLOCK except that reading of the record is allowed by other processes.

Any process that attempts a read or write on a locked record will sleep until the record is unlocked. If the record is locked with an RLOCK then reading is permissible. When a process terminates, all locked records are unlocked.

SEE ALSO

read (2), write (2), open (2)

DIAGNOSTICS

If an error occurs, -1 is returned. The error code EACCESS is returned if any portion of the record has been locked by another process for the LOCK & RLOCK actions. The error code EDEADLOCK is returned if locking the record would cause a deadlock. This error code is also returned if there are no more free internal locks.

savetty()	stored current tty flags
scanw(fmt,arg1,arg2,...)	scanf through *stdscr*
scroll(win)	scroll *win* one line
scrollok(win,boolf)	set scroll flag
setterm(name)	set term variables for name
unctrl(ch)	printable version of *ch*
waddch(win,ch)	add char to *win*
waddstr(win,str)	add string to *win*
wclear(win)	clear *win*
wclrtobot(win)	clear to bottom of *win*
wclrtoeol(win)	clear to end of line on *win*
werase(win)	erase *win*
wgetch(win)	get a char through *win*
wgetstr(win,str)	get a string through *win*
winch(win)	get char at current (y,x) in *win*
wmove(win,y,x)	set current (y,x) co-ordinates on *win*
wprintw(win,fmt,arg1,arg2,...)	printf on *win*
wrefresh(win)	make screen look like *win*
wscanw(win,fmt,arg1,arg2,...)	scanf through *win*

NAME

rdchk - check if there is data to be read

SYNOPSIS

```
rdchk(fdes);
int fdes;
```

DESCRIPTION

Rdchk checks to see if a process will block if it attempts to read the file designated by fdes. Rdchk returns 1 if there is data to be read or if it is the end of the file (EOF). In this context, the proper sequence of calls using rdchk is:

```
if (rdchk(fildes) > 0) read(fildes, buffer, nbytes);
```

SEE ALSO

read(2)

DIAGNOSTICS

Rdchk returns -1 if an error occurs (e.g., EBADF), 0 if the process will block if it issues a read and 1 if it is okay to read. EBADF is returned if a rdchk is done on a semaphore file or if the file specified doesn't exist.

for this prompt. These are bsh(1) commands and/or sh(1) commands.

An example menu for Electronic Mail Services is:

```
&Mail
|
!date                    \ELECTRONIC~MAIL~SERVICES
|
|         ~a - Receive~mail
|         ~b - Send~mail
|         ~c - Return~to~starting~menu
|
&Actions
~a      0
        mail
|
~b      -1
        echo -n "To whom do you wish to send mail?"
        read x
        echo "Now type the message."
        echo "Terminate it by typing a control -d."
        mail $x
~c
        Start
```

Body

&Menuidentifier must appear beginning in column one. Menuidentifier is any string having relevance to the user. A short descriptive string is usually best. The string may not contain any blanks or punctuation and it must begin with a capital letter. If the string ends with a question mark ("?"), the menu is called a "help menu." It will be invoked automatically when bsh is displaying the base menu and the user types a "?" command. Thus, the &Admin? menu is invoked when &Admin is the current menu and "?" is typed. The remainder of the &Menuidentifier line should be empty.

The body of each menu is composed of text which will be reproduced on the screen exactly as it appears (with exceptions as described below).

~prompt may occur one or more times within the body. This indicates a prompt for which there will be an associated action within the &Actions portion of the menu. Usually there will be a short phrase or sentence describing the action just to the right of the ~prompt. A prompt may be any letter, numeral or string of characters not containing punctuation. Usually shorter (1-2 character) prompts are preferred. A prompt must be separated from its surroundings by one

NAME

curses - screen functions with "optimal" cursor motion

SYNOPSIS

cc [flags] files -lcurses -ltermlib [libraries]

DESCRIPTION

These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, and the user sets up an image of a new one. Then the *refresh()* tells the routines to make the current screen look like the new one. In order to initialize the routines, the routine *initscr()* must be called before any of the other routines that deal with windows and screens are used.

SEE ALSO

Screen Updating and Cursor Movement Optimization: A Library Package, Ken Arnold.
termcap (5), stty (2), setenv (3), setenv (3).

AUTHOR

Ken Arnold

FUNCTIONS

addch(ch)	add a character to *stdscr*
addstr(str)	add a string to *stdscr*
box(win,vert,hor)	draw a box around a window
crmode()	set cbreak mode
clear()	clear *stdscr*
clearok(scr,bool)	set clear flag for *scr*
clrtobot()	clear to bottom on *stdscr*
clrtoeol()	clear to end of line on *stdscr*
delwin(win)	delete *win*
echo()	set echo mode
erase()	erase *stdscr*
getch()	get a char through *stdscr*
getstr(str)	get a string through *stdscr*
gettmode()	get tty modes
getyx(win,y,x)	get (y,x) co-ordinates
inch()	get char at current (y,x) co-ordinates
initscr()	initialize screens
leaveok(win,bool)	set leave flag for *win*
longname(termbuf,name)	get long name from *termbuf*
move(y,x)	move to (y,x) on *stdscr*
mvcur(lasty,lastx,newy,newx)	actually move cursor
newwin(lines,cols,begin_y,begin_x)	create a new window
nl()	set newline mapping
nocrmode()	unset cbreak mode
noecho()	unset echo mode
nonl()	unset newline mapping
noraw()	unset raw mode
overlay(win1,win2)	overlay win1 on win2
overwrite(win1,win2)	overwrite win1 on top of win2
printw(fmt,arg1,arg2,...)	printf on *stdscr*
raw()	set raw mode
refresh()	make current screen look like *stdscr*
resetty()	reset tty flags to stored value

The tilde "~" is an "escape character," and may not be used for any other pur-pose within a menu.

Each of the special escape sequences described above must be separated from surrounding text by one or more spaces or tabs.

It is important to know the number of lines and columns of the terminal(s) to be used with a menu system and to be certain not to create menus longer or wider than these values. Their values are specified within the termcap(5) file for each terminal upon which the Business Shell may be run.

Actions

&Actions must appear beginning in column one. &Actions must appear, even if there are no actions.

Each prompt in the actions section must be reproduced exactly as it appears in the body of the menu. It is the user's responsibility to ensure that the spelling of prompts in the body and actions sections match. The case of characters is significant; so "A" is not the same as "a."

Size is optional. It specifies the length of the window to be used during execution of the actions. If omitted, the default value is 5, and a window 5 rows by column columns will be reserved at the bottom of the screen for output. Column is the terminal column width as obtained from termcap(5). A size of Ø will reserve the entire screen. In this case, the screen is blanked prior to execution of the actions; and a prompt requesting a return or line feed is issued after execution. A negative size will reserve the entire screen similarly to the zero case, but after execution, the Business Shell is immediately resumed without waiting for a return or line feed. It is the user's responsibility to ensure that the action window is large enough.

The actions may be composed of bsh(1) commands or commands which are executed by the standard shell, sh(1). The actions should all be indented one tab stop from the left side of the file.

A bsh(1) command is the instruction to transfer immediately to a particular menu. This is specified by writing the name of the destination menu in the semantics field. Bsh(1) commands must be typed one-per-line.

Sh(1) commands follow the usual rules as described in Volume 1 of the **Programmer's Reference Manual.**

MENUS

menus -- format of a Business Shell menu system

DESCRIPTION

A menu system is a collection of menus which has been processed (digested) by digest(1M). The Business Shell, bsh(1), requires a menu system upon which to operate: it contains all the menus which are normally displayed to accomplish some set of functions. As distributed, the Business Shell includes the default menu system (/usr/lib/menusys and /etc/menusys.bin).

A menu source file may contain one or more individual menus. However, in the interest of maintainability, it is recommended that each menu source file contain only a single menu, or only very closely related menus. It is also recommended that the name of the menu source file and the menuidentifier be the same.

A source menu system may be a single menu file (containing one or more menus) or a directory structure containing menu files and subordinate directories.

Each menu file is an ASCII file consisting of two logical parts: the body and the actions. A (digested) menu system contains an additional part, the index. The index appears prior to the body. It specifies the byte-offset locations of each of the body and action sections as well as the associated menuidentifiers. Users should never attempt to construct an index by hand -- that is the function of digest(1M). Moreover, users should never attempt to edit a digested menu system; rather, the source menu files should be edited and then the menu system recreated using digest(1M).

The precise format of a menu source file is described below:

&Menuidentifier

The substance of the menu represented essentially as it is to be displayed. Within this area there usually will be one or more occurrences of:

~prompt strings

as well as other special commands as described below.

&Actions

Zero or more occurrences of:

~prompt size

The sequence of actions to be taken

or more spaces or tabs. If a menu name and a prompt both have the same spelling, the prompt is given preference in all cases.

!date inserts the current date and time, left-justified on the "!." The date/time format is "Tue Jul 13 17:10 1982." !date may appear more than once if desired.

!user inserts the current user id, left-justified on the "!." !user may appear more than once if desired.

!pwd inserts the current directory, left-justified on the "!." The full path name is displayed, e.g. /usr/jones/admin/currwork. !pwd may appear more than once if desired.

!@ indicates where the cursor is to be placed on the screen. Usually this should be just slightly to the right of the current prompt. If !@ is omitted, the cursor will be placed at the bottom left corner of the screen. At most, one occurrence of !@ should appear in each menu.

With the exception of !@, the "!" may appear as a suffix in which case the string will be right-justified instead of left-justified.

The "!" is an "escape characater", and may not be used for any other purpose within a menu.

\string denotes a string which is to be "highlighted" using the terminal's highlighting capabilities (usually reverse video). The "\" character must be on the left of the string. It is converted into the appropriate highlighting information during display. The string may be of any length up to the width of the display screen.

`string denotes a string which is to be "underlined" using the terminal's underlining capabilities (usually true underline). The "`" character must be on the left of the string. It is converted into the appropriate underlining information during display. The string may be of any length up to the width of the display screen.

The backslash "\" and backquote "`" as the initial letter of a string are "escape characters" and will always have the interpretations given above.

In order to create a highlighted or underlined string containing spaces, "significant spaces" may be represented as tildes ("~") within a string. Thus, \~hi~there~ will create a highlighted ten-character string.

ed	str		End delete mode
ei	str		End insert mode; give ":ei=:" if ic
eo	str		Can erase overstrikes with a blank
ff	str	(P*)	Hardcopy terminal page eject (default ^L)
hc	bool		Hardcopy terminal
hd	str		Half-line down (forward 1/2 linefeed)
ho	str		Home cursor (if no cm)
hu	str		Half-line up (reverse 1/2 linefeed)
hz	str		Hazeltine; can't print ~'s
ic	str	(P)	Insert character
if	str		Name of file containing is
im	bool		Insert mode (enter); give ":im=:" if ic
in	bool		Insert mode distinguishes nulls on display
ip	str	(P*)	Insert pad after character inserted
is	str		Terminal initialization string
k0-k9	str		Sent by "other" function keys 0-9
kb	str		Sent by backspace key
kd	str		Sent by terminal down arrow key
ke	str		Out of "keypad transmit" mode
kh	str		Sent by home key
kl	str		Sent by terminal left arrow key
kn	num		Number of "other" keys
ko	str		Termcap entries for other non-function keys
kr	str		Sent by terminal right arrow key
ks	str		Put terminal in "keypad transmit" mode
ku	str		Sent by terminal up arrow key
l0-l9	str		Labels on "other" function keys
li	num		Number of lines on screen or page
ll	str		Last line, first column (if no cm)
ma	str		Arrow key map, used by vi version 2 only
mi	bool		Safe to move while in insert mode
ml	str		Memory lock on above cursor.
mu	str		Memory unlock (turn off memory lock).
nc	bool		No correctly working carriage return (DM2500,H2000)
nd	str		Non-destructive space (cursor right)
nl	str	(P*)	Newline character (default \n)
ns	bool		Terminal is a CRT but doesn't scroll.
os	bool		Terminal overstrikes
pc	str		Pad character (rather than null)
pt	bool		Has hardware tabs (may need to be set with is)
se	str		End stand out mode
sf	str	(P)	Scroll forwards
sg	num		Number of blank chars left by so or se
so	str		Begin stand out mode
sr	str	(P)	Scroll reverse (backwards)
ta	str	(P)	Tab (other than ^I or with padding)
tc	str		Entry of similar terminal - must be last
te	str		String to end programs that use cm
ti	str		String to begin programs that use cm
uc	str		Underscore one char and move past it
ue	str		End underscore mode
ug	num		Number of blank chars left by us or ue

3-80

Since a menu file may contain one or more menus or directories containing menus, the recommended way to create a menu system is to create a tree of directories containing the various portions of the system. Each subtree contains all the menus related to a given subject. Thus, a primary menu (directory) is created for, say, system management functions: and subsidiary menus are placed beneath (within) the directory for each of the individual system management functions or function areas. Help menus may be placed wherever appropriate in the structure.

FILES

/usr/lib/menusys	source directory for /etc/menusys.bin
/etc/menusys.bin	digested default menu system for bsh(1)

SEE ALSO

bsh(1), digest (1M), sh(1), termcap(5)

as \072. If it is necessary to place a null character in a string capability it must be encoded as \200. The routines which deal with *termcap* use C strings, and strip the high bits of the output very late so that a \200 comes out as a \000 would.

Preparing Descriptions

We now outline how to prepare descriptions of terminals. The most effective way to prepare a terminal description is by imitating the description of a similar terminal in *termcap* and to build up a description gradually, using partial descriptions with *ex* to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the *termcap* file to describe it or bugs in *ex*. To easily test a new terminal description you can set the environment variable TERMCAP to a pathname of a file containing the description you are working on and the editor will look there rather than in */etc/termcap*. TERMCAP can also be set to the termcap entry itself to avoid reading the file when starting up the editor. (This only works on version 7 systems.)

Basic capabilities

The number of columns on each line for the terminal is given by the co numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the li capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the am capability. If the terminal can clear its screen, then this is given by the cl string capability. If the terminal can backspace, then it should have the bs capability, unless a backspace is accomplished by a character other than ^H (ugh) in which case you should give this character as the bc string capability. If it overstrikes (rather than clearing a position when a character is struck over) then it should have the os capability.

A very important point here is that the local cursor motions encoded in *termcap* are undefined at the left and top edges of a CRT terminal. The editor will never attempt to backspace around the left edge, nor will it attempt to go up locally off the top. The editor assumes that feeding off the bottom of the screen will cause the screen to scroll up, and the am capability tells whether the cursor sticks at the right edge of the screen. If the terminal has switch selectable automatic margins, the *termcap* file usually assumes that this is on, i.e. am.

These capabilities suffice to describe hardcopy and "glass-tty" terminals. Thus the model 33 teletype is described as

```
t3|33|tty33:co#72:os
```

while the Lear Siegler ADM-3 is described as

```
cl|adm3|3|lsi adm3:am:bs:cl=^Z:li#24:co#80
```

Cursor addressing

Cursor addressing in the terminal is described by a cm string capability, with *printf*(3s) like escapes %x in it. These substitute to encodings of the current line or column position, while other characters are passed through unchanged. If the cm string is thought of as being a function, then its arguments are the line and then the column to which motion is desired, and the % encodings have the following meanings:

%d	as in *printf*, 0 origin
%2	like %2d
%3	like %3d

3-82

NAME

termcap – terminal capability data base

SYNOPSIS

/etc/termcap

DESCRIPTION

Termcap is a data base describing terminals, used, *e.g.*, by *vi*(1) and *curses*(3). Terminals are described in *termcap* by giving a set of capabilities which they have, and by describing how operations are performed. Padding requirements and initialization sequences are included in *termcap*.

Entries in *termcap* consist of a number of ':' separated fields. The first entry for each terminal gives the names which are known for the terminal, separated by '|' characters. The first name is always 2 characters long and is used by older version 6 systems which store the terminal type in a 16 bit word in a systemwide data base. The second name given is the most common abbreviation for the terminal, and the last name given should be a long name fully identifying the terminal. The second name should contain no blanks; the last name may well contain blanks for readability.

CAPABILITIES

(P) indicates padding may be specified
(P*) indicates that padding may be based on no. lines affected

Name	Type	Pad?	Description
ae	str	(P)	End alternate character set
al	str	(P*)	Add new blank line
am	bool		Terminal has automatic margins
as	str	(P)	Start alternate character set
bc	str		Backspace if not ^H
bs	bool		Terminal can backspace with ^H
bt	str	(P)	Back tab
bw	bool		Backspace wraps from column 0 to last column
CC	str		Command character in prototype if terminal settable
cd	str	(P*)	Clear to end of display
ce	str	(P)	Clear to end of line
ch	str	(P)	Like cm but horizontal motion only, line stays same
cl	str	(P*)	Clear screen
cm	str	(P)	Cursor motion
co	num		Number of columns in a line
cr	str	(P*)	Carriage return, (default ^M)
cs	str	(P)	Change scrolling region (vt100), like cm
cv	str	(P)	Like ch but vertical only.
da	bool		Display may be retained above
dB	num		Number of millisec of bs delay needed
db	bool		Display may be retained below
dC	num		Number of millisec of cr delay needed
dc	str	(P*)	Delete character
dF	num		Number of millisec of ff delay needed
dl	str	(P*)	Delete line
dm	str		Delete mode (enter)
dN	num		Number of millisec of nl delay needed
do	str		Down one line
dT	num		Number of millisec of tab delay needed

There are two basic kinds of intelligent terminals with respect to insert/delete character which can be described using *termcap*. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can find out which kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type "abc def" using local cursor motions (not spaces) between the "abc" and the "def". Then position the cursor before the "abc" and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the "abc" shifts over to the "def" which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability in, which stands for "insert null". If your terminal does something different and unusual then you may have to modify the editor to get it to use the insert mode your terminal defines. We have seen no terminals which have an insert mode not not falling into one of these two classes.

The editor can handle both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as im the sequence to get into insert mode, or give it an empty value if your terminal uses a sequence to insert a blank position. Give as ei the sequence to leave insert mode (give this, with an empty value also if you gave im so). Now give as ic any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give ic, terminals which send a sequence to open a screen position should give it here. (Insert mode is preferable to the sequence to open a position on the screen if your terminal has both.) If post insert padding is needed, give this as a number of milliseconds in ip (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in ip.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g. if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability mi to speed up inserting in this case. Omitting mi will affect only speed. Some terminals (notably Datamedia's) must not have mi because of the way their insert mode works.

Finally, you can specify delete mode by giving dm and ed to enter and exit delete mode, and dc to delete a single character while in delete mode.

Highlighting, underlining, and visible bells

If your terminal has sequences to enter and exit standout mode these can be given as so and se respectively. If there are several flavors of standout mode (such as inverse video, blinking, or underlining – half bright is not usually an acceptable "standout" mode unless the terminal is in inverse video mode constantly) the preferred mode is inverse video by itself. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, this is acceptable, and although it may confuse some programs slightly, it can't be helped.

ul	bool	Terminal underlines even though it doesn't overstrike
up	str	Upline (cursor up)
us	str	Start underscore mode
vb	str	Visible bell (may not move cursor)
ve	str	Sequence to end open/visual mode
vs	str	Sequence to start open/visual mode
xb	bool	Beehive (f1=escape, f2=ctrl C)
xn	bool	A newline is ignored after a wrap (Concept)
xr	bool	Return acts like ce \r \n (Delta Data)
xs	bool	Standout not erased by writing over it (HP 264?)
xt	bool	Tabs are destructive, magic so char (Teleray 1061)

A Sample Entry

The following entry, which describes the Concept-100, is among the more complex entries in the *termcap* file as of this writing. (This particular concept entry is outdated, and is used as an example only.)

```
c1|c100|concept100:is=\EU\Ef\E7\E5\E8\E1\ENH\EK\E\200\Eo&\200:\
    :al=3*\E^R:am:bs:cd=16*\E^C:ce=16\E^S:cl=2*^L:cm=\Ea%+ %+ :co#80:\
    :dc=16\E^A:dl=3*\E^B:ei=\E\200:eo:im=\E^P:in:ip=16*:li#24:mi:nd=\E=:\
    :se=\Ed\Ee:so=\ED\EE:ta=8\t:ul:up=\E;:vb=\Ek\EK:xn:
```

Entries may continue onto multiple lines by giving a \ as the last character of a line, and that empty fields may be included for readability (here between the last field on a line and the first field on the next). Capabilities in *termcap* are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence which can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have two letter codes. For instance, the fact that the Concept has "automatic margins" (i.e. an automatic return and linefeed when the end of a line is reached) is indicated by the capability am. Hence the description of the Concept includes am. Numeric capabilities are followed by the character '#' and then the value. Thus co which indicates the number of columns the terminal has gives the value '80' for the Concept.

Finally, string valued capabilities, such as ce (clear to end of line sequence) are given by the two character code, an '=', and then a string ending at the next following ':'. A delay in milliseconds may appear after the '=' in such a capability, and padding characters are supplied by the editor after the remainder of the string is sent to provide this delay. The delay can be either a integer, e.g. '20', or an integer followed by an '*', i.e. '3*'. A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. When a '*' is specified, it is sometimes useful to give a delay of the form '3.5' specify a delay per unit to tenths of milliseconds.

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. A \E maps to an ESCAPE character, ^x maps to a control-x for any appropriate x, and the sequences \n \r \t \b \f give a newline, return, tab, backspace and formfeed. Finally, characters may be given as three octal digits after a \, and the characters ^ and \ may be given as \^ and \\. If it is necessary to place a : in a capability it must be escaped in octal

3-91

If tabs on the terminal require padding, or if the terminal uses a character other than ^I to tab, then this can be given as **ta**.

Hazeltine terminals, which don't allow '~' characters to be printed should indicate **hz**. Datamedia terminals, which echo carriage-return linefeed for carriage return and then ignore a following linefeed should indicate **nc**. Early Concept terminals, which ignore a linefeed immediately after an **am** wrap, should indicate **xn**. If an erase-eol is required to get rid of standout (instead of merely writing on top of it), **xs** should be given. Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt**. Other specific terminal problems may be corrected by adding more capabilities of the form **xx**.

Other capabilities include **is**, an initialization string for the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to properly clear and then set the tabs on the terminal, if the terminal has settable tabs. If both are given, **is** will be printed before **if**. This is useful where **if** is */usr/lib/tabset/std* but **is** clears the tabs first.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **tc** can be given with the name of the similar terminal. This capability must be *last* and the combined length of the two entries must not exceed 1024. Since *termlib* routines search the entry from left to right, and since the **tc** capability is replaced by the corresponding entry, the capabilities given at the left override the ones in the similar terminal. A capability can be cancelled with **xx@** where **xx** is the capability. For example, the entry

```
hn|2621nl:ks@:ke@:tc=2621:
```

defines a 2621nl that does not have the **ks** or **ke** capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

FILES

/etc/termcap file containing terminal descriptions

SEE ALSO

ex(1), curses(3), termcap(3), tset(1), vi(1), ul(1), more(1)

AUTHOR

William Joy
Mark Horton added underlining and keypad support

BUGS

Ex allows only 256 characters for string capabilities, and the routines in *termcap(3)* do not check for overflow of this buffer. The total length of a single entry (excluding only escaped newlines) may not exceed 1024.

The **ma**, **vs**, and **ve** entries are specific to the *vi* program.

Not all programs support all entries. There are entries that are not supported by any program.

%.	like %c
%+x	adds x to value, then %.
%>xy	if value > x adds y, no output.
%r	reverses order of line and column, no output
%i	increments line/column (for 1 origin)
%%	gives a single %
%n	exclusive or row and column with 0140 (DM2500)
%B	BCD (16*(x/10)) + (x%10), no output.
%D	Reverse coding (x-2*(x%16)), no output. (Delta Data).

Consider the HP2645, which, to get to row 3 and column 12, needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its cm capability is "cm=6\E&%r%2c%2Y". The Microterm ACT-IV needs the current row and column sent preceded by a ^T, with the row and column simply encoded in binary, "cm=^T%.%.". Terminals which use "%." need to be able to backspace the cursor (bs or bc), and to move the cursor up one line on the screen (up introduced below). This is necessary because it is not always safe to transmit \t, \n ^D and \r, as the system may change or discard them.

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus "cm=\E=%+ %+ ".

Cursor motions

If the terminal can move the cursor one position to the right, leaving the character at the current position unchanged, then this sequence should be given as nd (non-destructive space). If it can move the cursor up a line on the screen in the same column, this should be given as up. If the terminal has no cursor addressing capability, but can home the cursor (to very upper left corner of screen) then this can be given as ho; similarly a fast way of getting to the lower left hand corner can be given as ll; this may involve going up with up from the home position, but the editor will never do this itself (unless ll does) because it makes no assumption about the effect of moving up from the home position.

Area clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as ce. If the terminal can clear from the current position to the end of the display, then this should be given as cd. The editor only uses cd from the first column of a line.

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as al; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as dl; this is done only from the first position on the line to be deleted. If the terminal can scroll the screen backwards, then this can be given as sb, but just al suffices. If the terminal can retain display memory above then the da capability should be given; if display memory can be retained below then db should be given. These let the editor understand that deleting a line on the screen may bring non-blank lines up from below or that scrolling back with sb may bring down non-blank lines.

Insert/delete character

Codes to begin underlining and end underlining can be given as us and ue respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Microterm Mime, this can be given as uc. (If the underline code does not move the cursor to the right, give the code followed by a nondestructive space.)

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as vb; it must not move the cursor. If the terminal should be placed in a different mode during open and visual modes of ex, this can be given as vs and ve, sent at the start and end of these modes respectively. These can be used to change, e.g., from a underline to a block cursor and back.

If the terminal needs to be in a special mode when running a program that addresses the cursor, the codes to enter and exit this mode can be given as ti and te. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly.

If your terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike, then you should give the capability ul. If overstrikes are erasable with a blank, then this should be indicated by giving eo.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted HP 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as ks and ke. Otherwise the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as kl, kr, ku, kd, and kh respectively. If there are function keys such as f0, f1, ..., f9, the codes they send can be given as k0, k1, ..., k9. If these keys have labels other than the default f0 through f9, the labels can be given as l0, l1, ..., l9. If there are other keys that transmit the same code as the terminal expects for the corresponding function, such as clear screen, the *termcap* 2 letter codes can be given in the ko capability, for example, ":ko=cl,ll,sf,sb:", which says that the terminal has clear, home down, scroll down, and scroll up keys that transmit the same thing as the cl, ll, sf, and sb entries.

The ma entry is also used to indicate arrow keys on terminals which have single character arrow keys. It is obsolete but still in use in version 2 of vi, which must be run on some minicomputers due to memory limitations. This field is redundant with kl, kr, ku, kd, and kh. It consists of groups of two characters. In each group, the first character is what an arrow key sends, the second character is the corresponding vi command. These commands are h for kl, j for kd, k for ku, l for kr, and H for kh. For example, the mime would be :ma=^Kj^Zk^Xl: indicating arrow keys left (^H), down (^K), up (^Z), and right (^X). (There is no home key on the mime.)

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as pc.

NAME

ttytype - data base defining terminal type; used for associating terminals with serial ports

DESCRIPTION

The ttytype data base is used to associate a manufacturer's terminal with the different serial ports on the system. Each line contains the name of a terminal, a tab character, and then the XENIX device entry for the serial ports associated with that terminal. The terminal name must correspond to an entry in /etc/termcap.

Making an entry in the ttytype file for your terminals allows the system to make maximum use of terminal features for certain system facilities that use full screen capabilities. Among these programs are vi(1), bsh(1), and ua(1).

FILES

/etc/ttytype

SEE ALSO

vi(1), bsh(1), ua(1), termcap(5)

USAGE

A typical line in the ttytype file might look like "dumb /dev/tty3" or "wyse /dev/tty5." The first says that serial port 3 is connected to a terminal described in /etc/termcap as having no special characteristics such as cursor movement. The second entry tells XENIX that serial port 5 is connected to a terminal manufactured by Wyse Technology that is described in termcap under the name "wyse." The terminal name is the name found between the first and second vertical bars of the appropriate entry in /etc/termcap.

Appendix A

HARD DISK ORGANIZATION

CONFIGURATION

The built-in internal 10-megabyte hard disk on the 586 System is configured as follows:

CYLINDER	USE
0	Bootstrap program
1-40	Swap area
40-41	Alternate-sector area
42-end	XENIX file system

LOGICAL DEVICES

The following logical devices are defined in the Altos configuration of XENIX.

Logical Devices - Integral Hard Disk

LOGICAL	DEVICE	DESCRIPTION
0	hd0	all of hard disk (without sector mapping).
1	hd0a, swap	swap area.
2	hd0b, root	root file system.
3-8		unused.
9	hd0.spares	spare blocks for alternate sector mapping.
10		future expansion.
11	hd0.track0	all of track 0.
12	hd0.boot	primary bootstrap on track 0.
13	hd0.roc0	rest of cylinder 0. (Consists of cylinder 0 except for track 0.) Used for fsck temporary file.
14	hd0.layout	layout information. (See layout (1)).
15	hd0.secmap	mapping information for alternate sectors (See map (1)).

Appendix B

FLOPPY DISKETTE ORGANIZATION

CONFIGURATION

The floppy disk organization for a bootable XENIX File System is as follows:

TRACK	USE
Ø-54	Xenix file system
55-end	Swap area

LOGICAL DEVICES

The following logical devices are defined in the Altos configuration of XENIX:

LOGICAL DEVICE	TRACK	DEFINITION
fdØ	Ø-end	"Pseudo-tape" (see below) or file system
fdØ.swap	55-end	swap area

BOOTING FROM FLOPPY DISKETTE

XENIX is not setup to be run in multiuser mode after booting from the floppy diskette. Of course, it is fine to access a file system on the floppy diskette after booting off hard disk.

DISKETTES AS PSEUDO-TAPE

The floppy disk may be used sequentially as a "pseudo-tape", for example, by the tar utility. The command:

tar c file1 file2<CR>

archives file1 and file2 to the device /dev/tar, which is usually equivalent to the floppy disk device /dev/fd0

These files may be recovered from that diskette with the command:

tar x<CR>

For information on using this utility, see the section, "Saving and Restoring Files Using tar" in the **Altos Introduction to XENIX Manual.**

RANDOM ACCESS DISKETTE FILES

When diskettes are used with the tar utility, they are treated as sequential files. Files on those diskettes are read from beginning to end, as with tape files.

It is also possible to have "Random-Access" files on diskette. XENIX can use random access diskette files in the same way it uses the hard disk files. You can have additional files that you load into the system when needed and unload when not.

Initializing Diskettes as Random-Access Files

To use a diskette in this fashion, you must first initialize it with an empty file system as follows:

1. If necessary, format the diskette using the format utility. After the XENIX prompt, enter **format <cr>**, and follow the instructions given.

2. Insert (load) the formatted diskette.

3. Enter

   /etc/mkfs#/dev/fd0 1440<cr>

   Although the newly created file system is physically loaded, you must "mount" the file system before you can use it. Mounting gives XENIX the information to link the diskette with its own file system on your hard disk.

   Similarly, you must "dismount" (or unlink) the file system on the diskette before physically unloading that diskette.

Mounting a Diskette

Whenever a diskette file system is loaded, it must be mounted:

1. Insert (load) the diskette, if necessary.

2. Enter

   /etc/mount /dev/fd0 /fd<cr>

   You may now access the diskette's file system through the directory /fd.

   You can treat this directory as you would a directory on the hard disk. You can create files on it, transfer files to this directory, change or remove these files, etc.

Dismounting a Diskette

Before removing a mounted diskette, it must be dismounted:

1. Enter

   /etc/umount /dev/fd0<cr>

2. Remove (unload) the diskette from the drive.

As with tar diskettes, these diskette files should be labeled with a meaningful description and dated, and kept in a safe location when not being used.

APPENDIX C

SERIAL LINE PRINTER AND SPOOLER

STANDARD PRINTER CONFIGURATION

In the Altos implementation of XENIX, serial port 6 is configured for a serial printer operating at 9600 baud. The logical device name "/dev/lp" may be used to refer to this port, and the lpr utility references this device automatically for printing and spooling.

The lpr utility assumes that only one printer, /dev/lp, is attached to the system. If you want to connect more than one printer, refer to the "Connecting More Than One Printer" section of this appendix.

Printer spooling is a technique that mediates printer activity in a manner that allows all users of the system to share a printer without conflict. Files to be printed are copied to a spool directory (/usr/spool/lpd) and a background process moves those copies to the line printer device. This device is found in /dev, and is called "lp" (lp1, lp2, etc). Files in /dev are known as "special files", and are the interface to UNIX I/O. For an expanded discussion of special files in specific and I/O in general, see sections 29-32 of the **UNIX Programmer's Manual**, Volume 2B. A great deal of this material is specific to the PDP-11, however the mechanisms are the same as those for the Altos 586 computer system.

Any of several programs may be used to copy material to printer devices. For example:

```
cat /usr/john/doc > /dev/lp
```

This command copies the file "/usr/john/doc" directly to the default printer. If you have more than one printer, the default printer is the one that is most used. This has three possibly undesirable effects:

1. If /usr/john/doc is a big file, this command may take some time to complete.

2. If another user is copying a file to the printer at the same time, the result is probably not what anyone intended.

3. Since the cat program knows nothing about printers, and therefore nothing about baud rates, page sizes, margins, etc., the result may not be what is expected.

The lpr utility program is used to control printer requests. This program knows something about printers, how to set baud rates, etc.

To invoke the lpr utility, enter:

lprN [file_list]

Where: "N" is a digit from Ø to 5 and selects one of 6 printers.

Lpr may be invoked without entering a valve for "N" by entering:

lpr [file_list]

This command assumes the default printer, lp, and has the same effect as entering:

lprØ [file_list]

The lpr program copies the files in [file_list] to a spool directory and returns immediately to the invoker. Sometime later, perhaps up to 1Ø seconds, a printer (if not already busy) will begin printing. The printers themselves are physically connected to serial ports.

HARDWARE CONNECTIONS

The connection between the 586 computer system and a printer is a cable which has 25-pin subminiature D-type connectors. The computer's port hardware is "female", which requires that the computer side of a cable have a "male" connector. Most printers also have "female" port connectors; a compatible cable should have a "male" connector on each end. The most commonly used cables have at least pins 2, 3, and 2Ø connected from end to end.

Printer control of computer output is accomplished by either of two methods:

1. The printer should be configured to use the X-ON, X-OFF protocol, because XENIX uses this protocol to control the flow of data to the serial printer. This method requires that the printer send an X-OFF control code to the computer when overrun is about to occur. An X-ON control code is sent when it is safe for output to the printer to continue.

2. The second method controls the RS232C DTR (signal 20) signal to accomplish the same result. If you wish to use this method, be sure that the cable which connects the printer and the computer has this conductor.

CONNECTING MORE THAN ONE PRINTER

If you want to connect more than one printer, you should:

1. Log in as super-user (root).

2. You need to create appropriate device files in /dev. This is done with the ln command. First, select which printer is to be the default printer. This printer should be the most-used printer in the system.

To make the default printer device available for reassignment, enter:

mv /dev/lp /dev/olp

Next, select the port to which this printer is to be connected, by entering:

ln /dev/ttyP /dev/lp

Where: "P" is the port number of the serial port.

Next, configure the system for the additional printers to be supported, selecting which printer number (1-6) they are to be, and the number of the serial port which they ae to be connected, and enter:

ln /dev/ttyP /dev/lpN

Where: "P" is the serial port number, and "N" is the printer number. It is suggested that "P" and "N" be the same number to alleviate the confusion that occurs when printer 5 is connected to port 3.

Next, repeat the above ln command for each printer to be supported.

3. You need to make file names for invoking the lpr program. For each printer device file made in the previous step, enter:

ln /bin/lpr /bin/lprN

Where: "N" is a printer number.

4. You need to create spool directories. These directories are used to hold copies of material to be printed for each printer. For each printer device file made, enter:

mkdir /usr/spool/lpdN

Where: "N", as above, is a printer number.

NOTE: The default directory is already installed, do not try to create it. If any of the printers have baud rates other than 9600, refer to the next section "Changing/Setting Baud Rates".

CHANGING/SETTING BAUD RATES

If you want to change or set a terminal or printer to a a different baud rate than 9600, you should perform the following steps.

The /etc/ttys file contains entries of the form:

12ttyP

The above line is interpreted by various system programs. The first digit ("1" in the above example) tells the system to attempt to log on ttyP ("P" is a serial port number). The second digit specifies the baud rate for that particular terminal (see Baud Rate list below or GETTY(8) in volume I of the **UNIX Programmer's Manual** for the baud rates associated with these values.) For each printer to be supported, type a "disable" command for the corresponding serial port. This ensures that the system will not attempt to log on a port which is dedicated to a printer.

For example, if a printer is set up for port 6, enter:

disable tty6

Now, for each printer to be supported, add a line to file /etc/ttys that has the following format:

0BlpN

This line may be anywhere in the file. No spaces are permitted between portions of the line. "B" is a baud rate argument from the list below. "N" is the printer number.

If the default printer's baud rate is other than 9600, add:

0Blp

NOTE:

Printers whose baud rate is 9600 do not require a corresponding line in /etc/ttys.

Baud Rates

"B" VALUES	BAUD RATE
0	300
1	150
2	9600
3	1200
5	300
6	2400
7	4800

CONFIGURING SYSTEM WITHOUT A PRINTER

If you wish to support six terminals (with no printer), you should:

1. Log in as super-user (root).

2. Remove the lp entry in /dev by entering

   rm /dev/lp<CR>

3. Enable the login and shell on port 6 by editing the line referencing tty6 in the file /etc/ttys from "02tty6" to "12tty6", before going multiuser.

Appendix D

LIST OF TERMINAL CAPABILITIES

The basic XENIX system works with nearly all the generally available terminals, by making use of a standard "lowest common denominator" of terminal capabilities. However, some of the XENIX utilities, including many especially useful utilities, can make use of special terminal capabilites. A major example is the Business Shell.

For this reason, the /etc/termcap data base has been developed to describe terminal capabilities. The following pages give essential information extracted from /etc/termcap, in a form more easily understood that when the file itself is viewed. The information given describes all terminals currently supported for special use by the Altos release of the XENIX operating system.

Customizing Your Altos XENIX System

The following information explains how to inform XENIX of the special capabilities of the terminals you are using with the system.

The XENIX utilities, such as the Business Shell, that make use of special terminal capabilities access the file /etc/ttytype, which defines the type of the terminal attached to each serial port. It may be necessary to edit this file to provide the correct terminal type for each port. Each line in /etc/ttytype has two fields; the terminal type, and the associated port number.

The "terminal type" used in /etc/ttytype is the second field of the appropriate terminal entry in the /etc/termcap data base; that is, it is between the first two vertical bars, "|", in the entry. On the following list of "Terminals Supported by the

XENIX Operating System," the entry called "name of terminal" is the proper reference. When editing /etc/ttytype, use that name as the "terminal type." Any editor that is convenient can be used. Change the terminal type, if necessary, for each active serial port that your system uses. (See TTYTYPE(5) in the Utility Reference Section for more information about /etc/ttytype.)

For example, if you are using a TeleVideo terminal, current model 920, when you consult the following list you will find a group of entries for Televideo. The appropriate entry is "920b." Editing /etc/ttytype, you find that all ports are associated with "wyse." Change the port assignments you are using, or all port assignments, to "920b" and update the file.

If you do not find a listing for the terminal you are using, consult your dealer or Altos Customer Support.

Terminals Supported by the XENIX Operating System

The material below is derived from the system file/etc/termcap. This file describes terminal capabilities and characteristics. The use of this file is to support screen-oriented programs, such as vi. The /etc/termcap file is composed of a description entry for each supported terminal (and sometimes more than one, if the terminal has options, or is part of a family of products).

This document cites the name by which a particular terminal is known to the system, and contains a short description of the terminal, including the manufacturer's name, and other useful information. Included are comments relevant to the use of the terminal.

an example:

```
wyse                    WYSE WY-100
```

this entry indicates that the WYSE WY-100 terminal is supported by the system and that its name is 'wyse' (case is significant).

The 'name' of a terminal is specified to several system programs. Among them are:

the shell (sh):

```
% TERM = name; export TERM
```

the C shell (csh):

```
# setenv TERM name
```

or, for the default definition of a port.

a typical line in /etc/ttytype:

```
name    tty5
```

Please reference the appropriate documentation for an expanded explanation of the capabilities and uses of the above programs and structures.

Terminal naming conventions:

Terminal names look like:

<manufacturer> <model> - <modes/options>

Certain abreviations (e.g. c100 for concept100) are also allowed for upward compatibility. The part to the left of the dash, if a dash is present, describes the particular hardware of the teminal. The part to the right is used for flags indicating

special ROM's, extra memory, particular terminal modes, or user preferences. Names are always in lower case, for consistency in typing.

The following are conventionally used for flags:

rv Terminal in reverse video mode (black and white)
2p Has two pages of memory. Likewise 4p, 8p, etc.
w Wide - in 132 column mode.
pp Has a printer port which is used.
na No arrow keys - termcap ignores arrow keys which are actually there on the terminal, so the user can use the arrow keys locally.

Special manufacturer codes:

A: hardcopy daisy wheel terminals
M: Misc. (with only a few terminals)
q: Homemade
s: special (dialup, etc.)

the terminals:

NAME OF TERMINAL	DESCRIPTION
wyse	WYSE WY-100
du	dialup
hp	Hewlett-Packard
2621-nl	HP 2621 with no labels
2621	HP 2621
2621-wl	HP 2621 w/labels
hl9-u	Heathkit with underscore cursor
hl9-us	Heathkit w/keypad shifted/underscore cursor
hl9-bs	Heathkit w/keypad shifted
hl9	Heathkit hl9
cl00-rvs	slow reverse Concept 100
cl00-s	slow Concept 100
cl00-rvna	cl00 with no arrows
cl00-rvpp	cl00 with printer port
cl00	Concept 100
cl00-rv	cl00 rev video
adm3a	LSI ADM3A
adm3	LSI ADM3
mime	Microterm Mimel
bg	BBN BitGraph terminal
vt52	DEC VT52
gigi	DEC GIGI

A: DAISY WHEEL PRINTERS

The A manufacturer represents Diablo, DTC, Xerox, Qume and other Daisy wheel terminals.

1620	Diablo 1620
1620-m8	Diablo 1620 w/8 column left margin
dtc	DTC 382 with VDU
dtc300s	DTC 300s
gsi	
aj830	Anderson Jacobson
5520	NEC Spinwriter 5520
qume5	Qume Sprint 5
x1720	Xerox 1720 same as the Diablo 1620

C: CONTROL DATA

cdc456	CDC
cdc456tst	CDC

D: DATAMEDIA

dm1520	Datamedia 1520
dm2500	Datamedia 2500
dm3025	Datamedia 3025a
3045	Datamedia 3045a
dt80	Datamedia dt80/1
dt80w	Datamedia dt80/1 in 132 char mode

H: HAZELTINE

h1000	Hazeltine 1000
h1552	Hazeltine 1552
(be sure auto lf/cr switch is set to cr)	
h1552rv	Hazeltine 1552 reverse video
h1420	Hazeltine 1420
h1500	Hazeltine 1500
h1510	Hazeltine 1510
h1520	Hazeltine 1520
h2000	Hazeltine 2000

I: IBM, INTERACTIVE SYSTEMS, AND INTECOLOR

8001	ISC8001 Compucolor/Intecolor
compucolor2	CompucolorII
intest	Interactive Systems Corporation
(modified PE Owl 12000	
ibm	IBM 3101-10

M: MISCELLANEOUS TERMINALS

tab132	TAB 132/15
tab132w	TAB 132/15
tab132rv	TAB 132/15
tab132wrv	TAB 132/15

```
mw2                 Multiwriter 2
trs80               TRS-80 Radio Shack Model I
d800                Direct 800/A

vc404               Volker-Craig 404
vc404s              Volker-Craig 404 w/standout mode
vc404na             Volker-Craig 404 w/no arrow keys
vc404sna            Volker-Craig 404 w/standout mode and
                                     no arrow keys
vc303a              Volker-Craig 303A
vc303               Volker-Craig 303

ampex               Ampex Dialogue 80
dl32                Diagraphix 132a
soroc               Soroc 120

tec400              TEC scope
tec500              TEC 500
tec

teletec             Teletec Datascreen
digilog             Digilog 333
ep48                Execuport 4080
terminet1200        GE Terminet 1200
aed512              AED 512
datapoint           Datapoint 3360
falco               Falco TS-1
dg                  Data General 6053
cdi                 CDI 1203
     (^S is an arrow key. not recommended for use)
sol
x183                Cybernex XL-83
omron               Omron 8025AG
plasma              Plasma Panel
swtp                Southwest Technical Products CT82
terak               Terak emulating Datamedia 1520
virtual             CB unix virtual terminal
delta               Delta Data 5000
mdl110              Cybernex MDL-110
zen30               Zentec 30

N: ANN ARBOR

aa                  Ann Arbor 4080
aaa-18              Ann Arbor Ambassador/18 lines
aaa-20              Ann Arbor Ambassador/20 lines
aaa-22              Ann Arbor Ambassador/22 lines
aaa-24              Ann Arbor Ambassador/24 lines
aaa-26              Ann Arbor Ambassador/26 lines
aaa-28              Ann Arbor Ambassador/28 lines
aaa-30              Ann Arbor Ambassador/30 lines
aaa-36              Ann Arbor Ambassador/36 lines
aaa-40              Ann Arbor Ambassador/40 lines
aaa-48              Ann Arbor Ambassador/48 lines
```

```
aaa-60              Ann Arbor Ambassador/60 lines
aaa                 Ann Arbor Ambassador
aaa-db              Ann Arbor Ambassador 30
     (destructive backspace)

T: TELETYPE

33                  Model 33 Teletype
43                  Model 43 Teletype
37                  Model 37 Teletype

V: VISUAL

vi200               Visual 200 with function keys
vi200-rvic          Visual 200  revers video using insert
                    character
vi200-f             Visual 200 no function keys
vi200-rv            Visual 200 reverse video
vi200-ic            Visual 200 using insert character

X: TEKTRONIX

tek                 Tektronix 4012
tek4013             Tektronix 4013
tek4014             Tektronix 4014
tek4015             Tektronix 4015
tek4014-sm          Tektronix 4014 in small font
tek4015-sm          Tektronix 4015 in small font
tek4023             Tektronix 4023
4025                Tektronix 4024/4025/4027
4025-17             Tektronix 4025 17 line window
4025-17ws           Tektronix 4025 17 line window in
                      workspace
4025ex              Tektronix 4025 w/!

a: ADDS

Regent: lowest common denominator, works on all regents.

regent              ADDS Regent Series - works on all of
                      series.
regent100           ADDS Regent 100
regent20            ADDS Regent 20
regent25            ADDS Regent 25
regent40            ADDS Regent 40
regent60            ADDS Regent 60
regent60na          ADDS Regent 60 w/no arrow keys
a980                ADDS Consul 980
Note:  If return acts strangely on a980, check internal
switch 2 on the top chip on the Control PC board.
viewpoint           ADDS Consul 980
```

b: BEEHIVE

sb2	fixed Super Bee
bh3m	BeehiveIIIm
superbeeic	Super Bee with insert character
microb	Micro Bee series
sbl	Beehive Super Bee - fl=escape, f2=^C.

c: CONCEPT (HUMAN DESIGNED SYSTEMS)

There seem to be a number of different versions of the C108 PROMS. The first one that we had would lock out the keyboard if you sent lots of short lines (like /usr/dict/words) at 9600 baud. Try that on your C108 and see if it sends a ^S when you type it. If so, you have an old version of the PROMs. The old one also messed up vi with a 132-character line-length. You should configure the C108 to send ^S/^Q before running this.

c108	Concept 108 w/8 pages and ^S/^Q
c108	Concept 108 w/4 pages and ^S/^Q

Concepts have only window relative cursor addressing, not screen relative. To get it to work, a one page window scheme is used for screen style programs.

d: DEC (DIGITAL EQUIPMENT CORPORATION)

It is assumed that you have smooth scroll off or are at a slow enough baud rate that it doesn't matter (1200? or less). Also this assumes that you set auto-nl to on; if you set it off, use "vt100-nam."

The xon/off switch should be on.

vt100	DEC VT100
vt100	VT100 w/no am
gt42	DEC GT42
vt132	VT132
gt40	DEC GT40
vt50	DEC VT50
dwl	Decwriter I
vt50h	DEC VT50h
ovt100	old DEC VT100
vt100-s	DEC VT100 132 cols 14 lines w/o advanced video option)
vt100-w	DEC VT100 132 cols (w/advanced video)
dw2	Decwriter II
dw4	Decwriter IV

h: HEWLETT PACKARD

2621-A	2621 w/new ROM, strap A set
2621-45	HP 2621 with 45 keyboard
(should be used at 4800 baud or less)	
hp2645	HP 264x series
hp2626	HP 2626
(should be used at 4800 baud or less)	
hp2648	HP 2648a graphics terminal
2640	HP 2640a
2640b	HP 264x series
2621-48	HP 48 line 2621
2621-nt	HP 2621 w/no tabs
(2621 with no labels ever)	

i: INFOTON (GENERAL TERMINAL)

il00	General Terminal 100A
(formerly Infoton 100)	
i400	Infoton 400
addrinfo	
infotonKAS	

k: HEATHKIT (ZENITH)

h19-a	Heathkit H19 ANSI mode

l: LEAR SIEGLER (ADM)

If the adm31 gives you trouble with standout mode, check the DIP switch in position 6, bank @c11, 25% from back end of pc. Should be OFF. If there is no such switch, you have an old adm31 and must use oadm31.

adm31	LSI adm31
adm2	LSI adm2
adm42	LSI adm42
adm5	LSI adm5
adm3a+	ADM3A PLUS
oadm31	old ADM31

m: MICRTOTERM

These mimel entries refer to the Microterm Mime I or Mime II. The default mime is assumed to be in enhanced act iv mode.

```
mime3a              Mimel emulating 3a
microterm            Microterm Act IV
microterm5           Microterm Act V
act5s                skinny act5 - Act V in split screen mode
mime-fb              full bright Mimel
mime-hb              half bright Mimel
mime2a-s             Microterm Mime2a
     (emulating an enhanced Soroc IQ120)
     (but ^X can't be used a a kill character)
mime2a               Microterm Mime2a
     (emulating an enhanced vt52)
mime-3ax             Mimel emulating enhanced 3a

p: PERKIN ELMER

pe550                Perkin-Elmer 550
fox                  Perkin-Elmer 1100
owl                  Perkin-Elmer 1200

s: SPECIALS

Special "terminals'.  These are used to label tty lines when
you don't know what kind of terminal is on it.  The char-
acteristics of an unknwn terminal are the lowest common
denominator - they look about like a TI 700.

arpanet              network
bussiplexer
ethernet             network
lpr                  lineprinter
dumb                 unknown
switch               intelligent switch

t: TEXAS INSTRUMENTS

ti                   TI Silent 700
ti745                TI Silent 745
ti800                TI Omni 800

v: TELEVIDEO

Note: The 912 has a <funct> key that's like shift: <funct>8 xmits
"^A8/r".  The 920 has this plus real function keys that xmit
different things.  Termcap makes you use the funct key on the 912
but the real keys on the 920.

tvi912               TVI920 old TeleVideo
912b                 TVI new TeleVideo 912
920b                 TVI new TeleVideo 920
tvi912-2p            TeleVideo w/2 pages
                     set to page 1 when entering ex or vi.
                     reset to page 0 when exiting ex or vi.
tvi950-ap            TVI 950 w/alt pages
tvi950-b             bare TVI 950 no is
tvi950-ns            TVI 950 w/no standout
```

Note: The following TVI descriptions are for all 950's. It sets the following attributes:

full duplex
write protect off
conversation mode
graphics mode off
white on black
auto page flip off
turn off status line
clear status line
normal video
monitor mode off
edit mode
load blank character to space
line edit mode
enable buffer control
protect mode off
local edit keys
program unshifted send key to send line all
program shifted send key to send line unprotected

set the following to nulls:
field delimiter
line delimiter
start-protected field delimiter
end-protected field delimiter

set end of text delimiter to carriage return/null clear all column tabs

tvi950 TeleVideo 950

Note: tvi950 sets duplicate (send) edit keys (\El) when entering vi and sets local (no send) edit keys (\EK) when exiting vi

tvi950-2p TeleVideo 950 w/2 pages

Note: tvi950-2p is for 950 with two pages adds the following:

set 48 line page
place cursor at page 0, line 24, column 1
when entering ex or vi, set 24 line page
when exiting ex or vi, reset 48 line page,
place cursor at 0,24,1

tvi950-4p TeleVideo 950 w/4 pages

Note: tvi950-4p is for 950 with four pages adds the following:
set 96 line page
place cursor at page 0, line 24, column 1
when entering ex or vi, set 24 line page
when exiting ex or vi, reset 96 line page
place cursor at 0,24,1

tvi950-rv	TeleVideo 950 rev video
tvi950-rv2p	TeleVideo 950 rev video w/2 pages
tvi950-rv4p	TeleVideo 950 rev video w/4 pages

y: TELERAY

t3700	dumb Teleray 3700
t3800	Teleray 3800 series
tl061	Teleray 1061
tl061f	Teleray 1061 with fast PROM

Appendix E

NUMERIC FORMATS, C, AND FORTRAN 77

The following information is for reference only. This information on the internal formats used for numeric representation is not necessary for general use of the C language or Fortran 77. It can be useful when examining actual memory contents or doing other specialized system programming work.

The same formats are used by both languages.

INTEGER FORMATS

Integers and "short integers" are 16 bits in length. "Long integers" are 32 bits. For both sizes, the leftmost bit is a sign bit and the other 15 or 31 bits are magnitude. The sign is zero for positive, one for negative. Negative numbers are in twos-complement form.

The range of values is as follows:

Signand 15 bits	-32,768 to 32,767
Sign and 31 bits	-2,147,483,648 to 2,147,483,647

FLOATING-POINT FORMATS

Single precision floating point is 32 bits in length, double is 64. The leftmost eight bits consist of an exponent in excess 80 notation. "Excess 80" means that the hexadecimal values from 80 to FF are positive exponents, corresponding to 0 through 7F. Values less than 80 are negative exponents; 7F through 0 correspond to -1 through -7E.

The remaining 24 or 56 bits consist of a leading sign bit

and magnitude values. Magnitudes are normalized. "Normalized" means that the representation of magnitude and exponent is adjusted so that each magnitude value can be thought of as starting with .1nnn...

For example, the value of 101, decimal 5, would be .101 with an exponent of 3. The leftmost digit of magnitude does not need to be represented, because it is always 1 except for the special case of a value of zero. Therefore, the leftmost magnitude bit is not stored but is implied. It is referred to as the "hidden bit."

Example:

The value 15.25, decimal. In binary, this is 1111.01

(In binary, .1 = .5 decimal; .01 = .25, etc. Moving to the right of the point halves the value at each move, just as moving to the left of the point doubles the base 2 value.)

So, 1111.01 represents 15.25 decimal. Normalizing our binary value, we have .111101 with an exponent of 4. The exponent becomes 84 in excess 80 notation, or 1000 0100 in binary. The sign bit is zero (positive), and the magnitude is 11101000... with as many trailing zeros as needed. Notice that the leading ".1" has disappeared. It is the unnecessary "hidden bit." The binary and hexadecimal values are shown below.

```
1000 0100 0111 0100 0000 0000 0000 0000
  8    4  s 7    4    0    0    0    0
```

The example is single-precision. Double precision, in this case, would be the same with eight bytes (32 bits) of trailing zeros.

Other examples:

The fractional decimal value .625. In binary, this is .101; that is, .5 plus .125. The value is normalized as it is, the exponent is 0, the sign is positive, 0.

```
1000 0000 0010 0000 ...
  8    0  s 2    0  ...
```

Negative 5. In binary, 5 is 101. Before taking the twos-complement, we supply a leading zero which will become the negative sign bit: 0101. The twos-complement is 1011. Removing the sign bit, 011. Normalizing, .1100 with the exponent -2. In excess 80, -2 is 7E. Result:

```
0111 1110 1100 0000 ...
  7    E  s C    0  ...
```

Zero, the exception. This is an all zero value.

```
0000 0000 0000 0000 ...
  0    0    0    0  ...
```

All zeros can be thought of as zero by convention. Otherwise, it would represent the smallest positive number possible in the scheme.

VALUES IN MEMORY

As with other values in 8086 memory, floating point values are stored "back-words." The least signficant 16-bit word is stored first, then the next, and so forth. If the single-precision value 84740000 is stored at location x, it will show as follows when displaying memory contents:

```
x        0000
x + 2    8474
```

However, long integers are stored in order. The long integer with a hexadecimal value of 128A34BF will show as:

```
x        128A
x + 2    34BF
```

Appendix F

SAMPLE LIST OF UNIX UTILITIES

The following is a sample listing of the typical utilities provided in a full Xenix Development System.

You can obtain a list of your Xenix operating system's utilities by entering:

```
cd / <cr>
ls -FCR|lpr<cr>
```

LIST OF XENIX UTILITIES

bin	dev	install	priboot	xenix
boot	etc	lib	pribootfd	xenix.fd
boot.fd	fd	load.hd	tmp	
boot.fdhd	load.hd	lost+found	usr	

./bin:

ac	df	look	refer	test
adb	diff	lpr	restor	time
ar	du	ls	rev	tk
arcv	dump	m4	rm	touch
as	dumpdir	mail	rmail	tp
at	echo	make	rmdir	tr
awk	ed	man	sa	troff
basename	edit	mesg	sed	true
bc	esrep	mkdir	sh	tsort
bsh	enroll	mntchk	size	tty
cal	eqn	multiuser	sleep	uniq
calendar	ex	mv	sort	units
cat	expr	ncheck	sp	uucp
cb	f77	ndump	spell	uulog
cc	false	neqn	spline	uux
checkeq	fgrep	newgrp	split	v7grep
chgrp	file	nice	strip	v7login
chmod	find	nm	struct	v7ls
chown	flagbad	nroff	stty	v7ps
clri	fsck	od	su	vi
cmp	graph	osh	sum	vplot
col	grep	passwd	sync	vpr
comm	icheck	pr	t300	who
cp	join	prep	t300s	write
crypt	kill	prof	t450	xset
csh	l	ps	tabs	xsend
cu	ld	ptx	tail	yacc
date	learn	pwd	tar	yes

```
dc              lex             quot            tbl
dcheck          lint            random          tc
dd              ln              ranlib          tee
deroff          login           ratfor          tek

./dev:
altosnet        hd0.boot        lp              rhd0.secmap     tar
console         hd0.layout      mem             rhd0.spares     tty
cua0            hd0.roc0        null            rhd0.track0     tty2
cul0            hd0.secmap      rfd0            rhd0a           tty3
ether           hd0.spares      rfdl            rhd0b tty4
fd0             hd0.track0      rhd0            root tty5
fd0.swap        hd0a            rhd0.boot       rroot           tty6
fdl             hd0b            rhd0.layout     rswap
hd0             kmem            rhd0.roc0       swap

./etc:
accton          getty           menusys         newuser         ttys
asktime         group           mknod           rc              umount
checklist       haltsys         motd            shutdown        update
cron            inir            mount           systemid        utmp
ddate           init            mtab            termcap         wall

dial-login      menusys.bin     passwd
dmesg           mkfs            ttytype

./etc/newuser:

./fd:

./lib:
c0              f77cl           libI77.a        libm.a          libt4014.a
cl              f77c2           libc.a          libmp.a         libt450.a
c2              f77crt0.o       libcurses.a     libplot.a       libtermlib.
cpp             f77passl        libdbm.a        libt300.a       libunet.a
crt0.o          libF77.a        libln.a         libt300s.a      libvt0.a

./lost+found:

./tmp:

./usr:
adm             dict            lib             src             unix
altos           games           preserve        sys             user
bin             include         spool           tmp

./usr/adm:
acct            mssbuf          usracct
messages        savacct         wtmp

./usr/altos:
qa.text


./usr/bin:
```

Mail	double	last	plot	ua
apropos	enable	layout	print	ucp
chessclock	error	leave	printenv	ul
chfn	expand	lock	reset	users
chsh	fcopy	lookbib	script	uudecode
ckdir	ffmt	lorder	sddate	uuencode
clear	finser	makewhatis	see	uusend
clock	fleece	map	sendnet	uversion
copy	fmt	mkstr	settime	v7wc
ctags	fold	more	sizefs	w
cxref	format	msgs	soelim	wc
daytime	from	nohup	ssp	whatis
decode	setNAME	num	strings	whereis
diff3	sets	page	tod	whoami
disest	head	pcc	tra	whom
disable	iul	pconfig	tset	xstr

./usr/dict:

hlista	hstop	spellhist
hlistb	papers	words

./usr/dict/papers:

Ind.ia	Ind.ib	Ind.ic	Rv7man	runinv

./usr/games:

arithmetic	fish	master	random	wump
backgammon	fortune	number	snake	
banner	hangman	quiz	snscore	
craps	lib	quiz.k	ttt	

./usr/games/lib:

fortunes	mmhow	snake.log	snakerawscores

./usr/games/quiz.k:

africa	chinese	index	posneg	spell
america	collectives	latin	pres	state
areas	ed	locomotive	province	trek
arith	elements	midearth	seq-easy	ucc
asia	europe	morse	seq-hard	
babies	greek	murders	sexes	
bard	inca	poetry	sov	

./usr/include:

a.out.h	errno.h	olda.out.h	setty.h	time.h
ar.h	execargs.h	olddump.h	signal.h	tp-defs.h
assert.h	grp.h	pack.h	stddef.h	utmp.h
core.h	ident.h	psout.h	stdio.h	varargs.h
ctype.h	local	pwd.h	symbol.h	whoami.h
curses.h	math.h	regexp.h	sys	xout86.h
dk.h	mp.h	saio.h	sys.s	
dumprestor.h	mtab.h	setjmp.h	sysexits.h	

./usr/include/local:

layout.h	sspare.h	uparm.h

```
./usr/includes/sys:
acct.h          file.h          mount.h         proc.h          timeb.h
buf.h           filsys.h        mpx.h           res.h           times.h
callo.h         ino.h           mx.h            sc.h            tty.h
chars.h         inode.h         param.h         sites.h         types.h
conf.h          ioctl.h         pk.h            stat.h          user.h
dir.h           locking.h       pk.p            systm.h
fblk.h          map.h           prim.h          text.h

./usr/lib:
Mail.help       crontab         font            llib-port       tabset
Mail.help.~     crontab.noUNET  learn           lpd             term
atrun           diff3           lex             me              tmac
bsh             ex2.13reserve   lintl           menusys         uucp
bsh.messages    ex2.13recover   lint2           more.help       yaccpar
calendar        ex2.13strings   llib-lc         refer
cign            ffmt            llib-lm         struct

./usr/lib/font:
ftB     ftCE    ftCS    ftGI    ftI     ftPA    ftR     ftSI    ftXM
ftBC    ftCI    ftCW    ftGM    ftL     ftPB    ftS     ftSM
ftC     ftCK    ftG     ftGR    ftLI    ftPI    ftSB    ftUD

./usr/lib/learn:
C.a             Xinfo           files.a         makefile
Linfo           editor.a        lcount          morefiles.a
READ_ME         eqn.a           macros.a        tee

./usr/lib/lex:
ncform

./usr/lib/me:
acm.me          eqn.me          index.me        revisions       thesis.me
chars.me        float.me        local.me        sh.me
deltext.me      footnote.me     null.me         tbl.me

./usr/lib/menusys:
Backup          Dir             Execute?        Mail            Start?
Backup?         Dir?            Help            Mail?           SysAdmin
Commands?       Execute         Help?           Start           SysAdmin?

./usr/lib/refer:
hunt            inv             mkey

./usr/lib/struct:
beautify        structure

./usr/lib/tabset:
beehive         std             vt100
diablo          teleray         xerox1720

./usr/lib/term:
tab300          tab300s-12      tab450-12       tabal
tab300-12       tab37           tab450-12-8     tablp
tab300s         tab450          tab832          tabn300
```

```
./usr/lib/tmac:
tmac.an         tmac.help     tmac.s          tmac.sdisp      tmac.sref
tmac.e          tmac.r        tmac.scover     tmac.skeep

./usr/lib/uucp:
L-devices       L.sys         uucico          uuxqt
L-dialcodes     USERFILE      uuclean

./usr/preserve:

./usr/spool:
at              mail          tunetmail       uucp
lpd             msgs          unetmail        uucppublic

./usr/spool/at:
lasttimedone    past

./usr/spool/at/past:

./usr/spool/lpd:

./user/spool/mail:

./usr/spool/msgs:
bounds

./usr/spool/uucp:

./usr/spool/uucppublic:

./usr/src:
cmd

./usr/srs/cmd:
decode.c

./usr/sys:

./usr/tmp:

./usr/unix:

./usr/user:
```

Appendix G

COPYING FILES FROM THE ALTOS 8600 TO THE ALTOS 586 UNDER THE XENIX OPERATING SYSTEM

If you want to transfer files from an Altos ACS 8600 to an Altos 586 system, the best method is through a uucp network. Bell Labs developed the uucp group of programs to facilitate the regular transfer of files between systems using the UNIX operating system. (Uucp stands for Unix-to-Unix Copy.) This appendix describes how to use uucp for a different purpose: The one-time transfer of a large number of files from an 8600 to a 586. Two assumptions are made here about your needs:

-- It's assumed that you don't want to regularly transfer files.

-- It's assumed that the two systems can be placed together so they can be directly hooked up.

If these assumptions don't match your needs, then you should turn to the description of uucp networks in the UNIX Programmer's Manual that came with your XENIX operating system. You can find complete documentation of these networks there. This document describes only those features of uucp needed for a one-time transfer.

Both systems must be using the XENIX Development System with the uucp program installed.

The information in this appendix is organized into four major sections:

1. Connecting the 8600 and the 586
2. Preparing the Configuration Files
3. Disabling and Enabling the TTY Ports
4. Testing the Uucp Network
5. Copying Files Using Uucp

It's assumed that you are familiar with the XENIX operating sytem and its major features. It's also assumed that you know how to use at least one of the XENIX editors. If you need more information on either Xenix or its editors, refer to the UNIX Programmer's Manual for more information.

CONNECTING THE ACS 8600 AND THE 586

The 8600 and 586 systems should be placed close enough together that they can be directly connected by a single null-modem cable. You can connect the cable to any port on the two systems that isn't the port used by the system terminal on that system. You can have any arrangement of peripheral devices attached to either system so long as both systems at least have a system terminal connected to the them.

NOTE

The systems must be connected using a null-modem cable for the procedure to work.

We suggest that you connect the two systems through their tty5 ports. The examples in this document show the systems connected through these ports. If you connect the systems through other ports, be sure to modify the examples to reflect your setup.

Also, ensure that both systems are set up for multiple users. If either system is in a single-user mode, lop in as super-user and type in

multiuser <cr>

PREPARING THE CONFIGURATION FILES

The uucp program comes ready to use. It does need, however, certain information to establish the connection between the 586 and 8600 systems. You provide this information by adding entries to several files on each system. The following table gives the steps needed for each system to prepare the files:

TASK:	FILE EFFECTED:
Assign a system name to the system	/etc/systemid
Define the communications line characteristics	/usr/lib/uucp/L-devices
Give information needed to login to the other system	/usr/lib/uucp/L.sys
Specify file accessibility	/usr/lib/uucp/USERFILE

Unless you have special requirements, you probably can edit the files on both systems in a few minutes. To make the task simpler, this section gives recommended entries. Some versions of the XENIX that comes with the 586 already have the recommended entries placed in the files. In this case, you don't have to add anything to the 586 files, but must still modify the 8600 files.

You can use the XENIX editor to check the contents of the 586's files to see if you must modify them.

In case you have some special requirements, this document also describes how to prepare your own entries.

You'll use one of the XENIX editors to add the entries to the files. To edit the files, you must be a XENIX superuser (root). You can become a superuser either by logging in as root or by using the su command.

Recommended Entries

You can use a set of standard entries to set up the 586's files if your requirements meet these assumptions:

1. You must assign the system name **Altos86** to the 8600 system and the name **Altos586** to the 586 system. If you don't, you must give different system names in the /usr/lib/uucp/L.sys and /usr/lib/uucp/USERFILE files.

2. The line connecting the two systems must connect into port tty5 on the each system. If it doesn't, you must give different port names in the /usr/lib/uucp/L-devices and /usr/lib/uucp/L.sys files.

3. The connection between the two systems must be direct. That is, it can't go through a telephone system. If it isn't a direct connection, you must give a different baud rate in the /usr/lib/uucp/L-devices and /usr/lib/uucp/L.sys files.

If your requirements don't meet these assumptions, read the instructions in the section "If You Have Special Requirements." They tell you how to tailor the file entries to yor requirements. If your requirements do match these assumptions, copy these entries into the files shown if they are not already there:

FOR THE 586:

FILE	ENTRY
/etc/systemid	Altos586
/usr/lib/uucp/L-devices	tty5 0 9600
/usr/lib/uucp/L.sys	Altos86 Any tty5 9600 tty5 ogin:-^M-ogin:-^M-ogin:uucp
/usr/lib/uucp/USERFILE	root, / , /usr /tmp

The entry for the /usr/lib/uucp/L.sys file must have the carriage

returns (^M) embedded as shown. See the UNIX manuals for information on how to embed carriage returns within a character string using your editor.

FOR THE 8600:

FILE	ENTRY
/etc/systemid	Altos86
/usr/lib/uucp/L-devices	tty5 0 9600
/usr/lib/uucp/L.sys	Altos586 Never tty5 9600 tty5
/usr/lib/uucp/USERFILE	root, / , /usr /tmp

If these recommended entries meet your needs, skip the next section and go to the section "Testing the Uucp Network."

IF YOU HAVE SPECIAL REQUIREMENTS

If you can't use the suggested entries, the following subsections give instructions on preparing each file. This section is organized as follows:

-- Assigning System Names

-- Defining the Communications Line Characteristics

-- Supplying the Login Information

-- Defining the File Accessibility

Assigning the System Names

Uucp needs a unique name for each system. The names identify each system in commands and during the login. To assign a system name, use an editor to add a line to the file **/etc/systemid.** This line should contain a single word entry that can be any legal UNIX name. The name cannot be the same name as any other system name that this system will communicate with through uucp. The /etc/systemid file can contain more than one system name each. Any name in this file can be used with uucp, but we suggest that you use just one name per system to avoid confusion.

Defining the Communications Line Characteristics

Uucp needs certain information about the communications line it will use. To provide this information, edit the file **/usr/lib/uucp/L-devices** on each system to add a line of this format:

format for both systems:

port call-unit baud-rate

where:

port	names the port to be used.
call-unit	Enter a **0** (zero) for this field.
baud-rate	gives the baud rate of the line. If the systems are directly connected, the baud rate is 9600.

This entry:

tty5 0 9600

states that the line connects through port tty5 and has a baud rate of 9600.

If the communications line can operate at more than one baud rate, you must include a separate entry for each baud rate as done here:

tty5 0 300
tty5 0 600

Supplying the Login Information

Uucp needs certain information to establish a connection between the systems. To provide this information, edit the file **/usr/lib/uucp/L.sys** to add a line of this format:

format for the 586 system:

system-name time port baud-rate phone login

format for the 8600 system:

system-name time port baud-rate phone

where:

system-name	gives the name assigned to the other system in it's /etc/systemid file.
time	gives the times that the uucp program is to try to login to the other system. For 586 system, state **Any.** This has uucp establish the connection any time you call it. For the 8600 system, state **Never.** This prevents the 8600 from ever making the connection.

port — names the port through which the connection is made to the other system. The port name must match the port name given in the system's /usr/lib/uucp/L-devices file.

baud-rate — gives the baud rate that is to be used. The baud rate must match one of the baud rates given for the port in the system's /usr/lib/uucp/L-devices file.

phone — must be the same name given for the port field of this entry.

login — for the 586 only, consists of a series of fields telling uucp how to login to the 8600 system. The entry should be:

```
ogin:-^M-ogin:-^M-ogin: uucp
```

The **^M** characters in the string are carriage returns (CONTROL-M) embedded with the string. These carriage returns must appear within the file as shown. See the UNIX documentation for information on how to embed control characters within strings using your editor.

Defining the File Accessibility

Uucp needs permission to access files on either system. To provide permission, edit the file **/usr/lib/uucp/USERFILE** on each system to lines of this format:

format for both systems:

```
root,   /
,  /usr /tmp
```

where:

root, / — gives the superuser on either system access to any file in any directory through uucp.

, /usr /tmp — gives any non-superuser on either system access to any file in any daughter directory of the /usr /tmp directories through uucp.

DISABLING AND ENABLING THE TTY PORTS

Before testing the uucp network and copying files using uucp, the following steps must be performed:

1. On the 586, enter:

disable /dev/tty5

Substitute the name of the pot you're using in this command if the connection to the 8600 isn't through port tty5.

2. On the 8600, enter:

enable /dev/tty5

Substitute the name of the port you're using in this command.

TESTING THE UUCP NETWORK

Before you begin copying files from the 8600 to the 586, you should test the network by copying a single file. If the copy succeeds, you can start copying over the bulk of your files. If it doesn't succeed, you must check your connection and your configuration files.

The test copies the file /etc/passwd from the 586 to the the file /tmp/passwd on the 8600. To conduct the test, follow these steps:

1. Boot and become a superuser (root) on both systems.

2. On the 586, enter:

uucp /etc/passwd Altos86\!/tmp/passwd

Substitute the system name you gave the 8600 in this command if you didn't name it Altos8600 in its /etc/systemid file.

3. The copy takes about one minute to complete. After that time, on the 8600, enter:

cat /tmp/passwd

If cat shows that the file /tmp/passwd contains the contents of the file /etc/passwd on the 586, then the uucp copy worked. If the /tmp/passwd file doesn't exist or is empty, then the copy didn't work.

If the copy works, then go on to the section "Copying Files Using Uucp." If the copy didn't work, check the connection between the two systems. Once you're sure that the cable is properly connected (and that nothing is wrong with the cable) try the steps above again. If they still don't work, check the contents of the configuration files you prepared. Once you're sure that they are correct, again try the copy.

If you still have problems, use the information below to try to debug your setup. These steps describe what happens when uucp performs a copy. By looking at the files mentioned, you should be able to determine where the problem lies. Then turn to the **UNIX Programmer's Manual.** It contains more information on uucp

that should be helpful for solving your problem.

When uucp performs the copy, these steps should occur:

1. The uucp program creates two files in the 586's /usr/spool/uucp directory. The first, D.Altos8600n0001, contains a copy of the file /etc/passwd. The second file, C.Altos8600n0001, contains control information. (The names of these files will be different if you didn't assign the name Altos86 to the 8600.)

   Uucp also places the message, "QUEUED (C.Altos8600n0001)" in the file /usr/spool/uucp/LOGFILE on the 586.

   At the end of this step, the program uucp stops execution.

   If a file /usr/spool/uucp/STST* exists on the 586, remove it before retrying the procedure.

2. The program uucico then begins execution. It's first task is to examine the 586 file /usr/lib/uucp/L.sys. The entry in the file tells uucico to immediately login to the 8600. The following steps occur as part of the login:

   -- Uucico sends a carriage return to the 8600, which should respond with login message. Uucico then logs in on the 8600.

   -- The uucico program on the 586 executes the uucico program on the 8600.

   -- The uucico program on the 586 creates two temporary files in the 586's /usr/spool/uucp directory that are prefixed with "LCK".

   -- Uucico on the 586 places the message "SUCCEEDED (call to Altos86)" in the 586 file /usr/spool/uucp/LOGFILE.

3. The uucico program on the 586 checks its spool directory and learns that it should transfer a file from the 586 to the 8600. The message "REQUEST (S /etc/passwd /tmp/passwd username) is placed in the /usr/spool/uucp/LOGFILE files on both systems.

4. Uucico on the 586 transfers the file D.Altos8600n0001, which is a copy of /etc/passwd, from the 586 to the 8600. The uucico program on the 8600 places the file in the directory /usr/spool/uucp. It then moves the file to the file /tmp/passwd.

5. The message "REQUEST (SUCCEEDED)" is placed in the /usr/spool/uucp/LOGFILE files on both systems.

COPYING FILES USING UUCP

After you've tested the connection and the configuration files, you can begin copying files from the 8600 to the 586. Follow these steps to do the copying:

1. Turn on and boot both systems. Log in as the superuser on both systems.

2. If any of the 8600 files you want to copy aren't part of the 8600 directories, copy them into a directory. (These typically would be files that you've copied onto a diskette or tape using the tar command.)

3. Use the uucp command on the 586 to copy files from the 8600 to the 586. The last section in this appendix, "Using the Uucp Command," gives instructions on using the uucp command. You can use the uucp command as many times as necessary to copy files.

USING THE UUCP COMMAND

Once you've enabled and disabled the ports, you can begin using uucp to copy files. The basic format of the uucp command is:

uucp [-d] 86-system-name!source-file destination-file

where:

-d is an optional parameter that has uucp create, if necessary, all necessary directories to place the source file(s) in the destination file given

8600-system-name

gives the name you assigned to the 8600 in its /etc/systemid file. You must follow the system name with an exclamation mark (!).

source-file gives the name of the source file or files to be copied from the 8600. The name must include the pathname to the directory that contains the file or files. The name can include the metacharaters **? * []** that the 8600 will expand. Uucp will copy every file that whose name fits in the expanded name.

destination-file

gives the name of the file into which uucp will place the contents of the source file. If a pathname is given, uucp places the copied file into the named directory.

Otherwise, the copied file goes into the current directory. If more than one file is copied, then the copied files are placed into files of the same name as the files on the 8600 system.

Let's say that you want to copy the entire contents of the directory /usr/marketing/reports from the 8600 to a directory of the same name on the 586. You would use this command:

uucp -d Altos86!/usr/marketing/reports/* /usr/marketing/reports

The asterick (*) following the Altos8600 pathname has uucp copy all the files in the directory. The -d has uucp create the directory /usr/marketing/reports on the 586 if it doesn't already exist. (Note that in this example, the 8600 has the system name Altos8600. In your commands, you would substitute the name you assigned the 8600.)

In the next example, let's say that you want to copy the file y_t_d_sales into the current directory on the 586. You would use this command:

uucp Altos86!/usr/jane/sales/y_t_d_sales

This has uucp place the file into the current directory in a file of the same name as on the 8600.

Appendix H

8086 ASSEMBLY LANGUAGE REFERENCE MANUAL

The following pages represent an 8086 Assembly Language Reference Manual extracted with permission from a Microsoft, Inc. publication. The section and page numbers of this excerpt reflect the enumeration and pagination of the original publication.

2.5 AS: The XENIX Assembler

This document describes the usage and input syntax of the XENIX 8086 assembler as. As is an assembler that produces an output file containing relocation information and a complete symbol table. The output is acceptable to the XENIX loader ld, which may be used to combine the outputs of several assembler runs and to obtain object programs from libraries. The output format has been designed so that if a program contains no unresolved references to external symbols, it is executable without further processing.

2.5.1 Usage

As is invoked as follows:

as [-l] [-o output] file

If the optional `-l' argument is given, an assembly listing is produced which includes the source, the assembled (binary) code, and any assembly errors.

The output of the assembler is by default placed on the file a86.out in the current directory; The `-o' flag causes the output to be placed on the named file.

2.5.2 Lexical conventions

Assembler tokens include identifiers (alternatively, ``symbols'' or ``names''), constants, and operators.

2.5.2.1 Identifiers An identifier consists of a sequence of alphanumeric characters (including period ``.''and underscore ``_'' as alphanumeric) of which the first may not be numeric. Only the first eight characters are significant. The case of alphabetics in identifiers is significant.

2.5.2.2 Constants A hex constant consists of a sequence of digits and the letters `a', `b', `c', `d', `e', and `f' (any of which may be capitalized), preceeded by the character `/'. The letters are interpreted with the following values:

HEX	DECIMAL
A	10
B	11
C	12
D	13
E	14
F	15

An octal constant consists of a series of digits, preceded by the tilde character `` ''. The digits must be in the range from 0 to 7.

A decimal constant consists simply of a sequence of digits. The magnitude of the constant should be representable in 15 bits; i.e., be less than 32,768.

2.5.2.3 Blanks Blank and tab characters may be freely interspersed between tokens, but may not be used within tokens (except in character constants). A blank or tab is required to separate adjacent identifiers or constants not otherwise separated.

2.5.2.4 Comments The character ``|'' introduces a comment, which extends through the end of the line on which it appears. Comments are ignored by the assembler.

2.5.3 Segments

Assembled code and data fall into three segments: the text segment, the data segment, and the bss segment. The text segment is the one in which the assembly begins, and it is the one into which instructions are typically placed. The XENIX system will, if desired, enforce the purity of the text segment of programs by trapping write operations into it. Object programs produced by the assembler must be processed by the link-editor ld (using its `-i' flag) if the text segment is to be write-protected. A single copy of the text segment is shared among all processes executing such a program.

The data segment is available for placing data or instructions which will be modified during execution. Anything which may go in the text segment may be put into the data segment. In programs with write-protected, sharable text segments, the data segment contains the initialized but variable parts of a program. If the text segment is not pure, the data segment begins immediately after the text segment. If the text segment is pure, the

data segment is in an address space of its own, starting at location zero (0).

The bss segment may not contain any explicitly initialized code or data. The length of the bss segment (like that of text or data) is determined by the high-water mark of the location counter within it. The bss segment is actually an extension of the data segment and begins immediately after it. At the start of execution of a program, the bss segment is set to 0. The advantage in using the bss segment for storage that starts off empty is that the initialization information need not be stored in the output file. See also location counter and assignment statements below.

2.5.4 The location counter

The special symbol, ``.'', is the location counter. Its value at any time is the offset within the appropriate segment from the start of the statement in which it appears. The location counter may be assigned to, with the restriction that the current segment may not change; furthermore, the value of ``.'' may not decrease. If the effect of the assignment is to increase the value of ``.'', the required number of null bytes are generated (but see Segments above).

2.5.5 Statements

A source program is composed of a sequence of statements. Statements are separated by new-lines. There are four kinds of statements: null statements, expression statements, assignment statements, and keyword statements.

The format for most 8086 assembly language source statements is:

```
[<label field>]
op-code [<operand field>] [<comment>]
```

Any kind of statement may be preceded by one or more labels.

2.5.5.1 Labels There are two kinds of labels: name labels and numeric labels. A name label consists of a identifier followed by a colon (:). The effect of a name label is to assign the current value and type of the location counter ``.'' to the name. An error is indicated in pass 1 if the name is already defined; an error is indicated in pass 2 if the ``.'' value assigned changes the definition of the

label.

A numeric label consists of a string of digits 0 to 9 and dollar-sign ($) followed by a colon (:). Such a label serves to define local symbols of the form ``n$'', where n is the digit of the label. The scope of the numeric label is the labelled block in which it appears. As an example, the label 9$ is defined only between the lables foobar and foo:

```
foobar:
9$:     .byte 0
        .
        .
        .
foo:    .word a
```

As in the case of name labels, a numeric label assigns the current value and type of ``.'' to the symbol.

2.5.5.2 Null statements A null statement is an empty statement (which may, however, have labels and a comment). A null statement is ignored by the assembler. Common examples of null statements are empty lines or lines containing only a label.

2.5.5.3 Expression statements An expression statement consists of an arithmetic expression not beginning with a keyword. The assembler computes its value and places it in the output stream, together with the appropriate relocation bits.

2.5.5.4 Assignment statements An assignment statement consists of an identifier, an equal sign (=), and an expression. The value and type of the expression are assigned to the identifier. It is not required that the type or value be the same in pass 2 as in pass 1, nor is it an error to redefine any symbol by assignment.

Any external attribute of the expression is lost across an assignment. This means that it is not possible to declare a global symbol by assigning to it, and that it is impossible to define a symbol to be offset from a non-locally defined global symbol.

As mentioned, it is permissible to assign to the location counter ``.''. It is required, however, that the type of the expression assigned be of the same type as ``.'', and it is forbidden to decrease the value of ``.''. In practice,

the most common assignment to ``.'' has the form ``.=.+n'' for some number n; this has the effect of generating n null bytes.

2.5.5.5 Keyword statements Keyword statements are numerically the most common type, since most machine instructions are of this sort. A keyword statement begins with one of the many predefined keywords of the assembler; the syntax of the remainder depends on the keyword. All the keywords are listed below with the syntax they require.

2.5.6 Expressions

An expression is a sequence of symbols representing a value. Its constituents are identifiers, constants, and operators. Each expression has a type.

Arithmetic is two's complement. All operators have equal precedence, and expressions are evaluated strictly left to right.

2.5.6.1 Expression operators The operators are:

Operator	Description
(blank)	same as +
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Logical OR
&	Logical AND
!	Logical NOT
>	Right Shift
<	Left Shift

2.5.6.2 Types The assembler deals with expressions, each of which may be of a different type. Most types are attached to the keywords and are used to select the routine which treats that keyword. The types likely to be met explicitly are:

undefined
: Upon first encounter, each symbol is undefined. It may become undefined if it is assigned an undefined expression.

undefined external

A symbol which is declared .globl but not defined in the current assembly is an undefined external. If such a symbol is declared, the link editor ld must be used to load the assembler's output with another routine that defines the undefined reference.

absolute

An absolute symbol is defined ultimately from a constant. Its value is unaffected by any possible future applications of the link-editor to the output file.

text

The value of a text symbol is measured with respect to the beginning of the text segment of the program. If the assembler output is link-edited, its text symbols may change in value since the program need not be the first in the link editor's output. Most text symbols are defined by appearing as labels. At the start of an assembly, the value of ``.'' is text 0.

data

The value of a data symbol is measured with respect to the origin of the data segment of a program. Like text symbols, the value of a data symbol may change during a subsequent link-editor run since previously loaded programs may have data segments. After the first .data statement, the value of ``.'' is data 0.

bss

The value of a bss symbol is measured from the beginning of the bss segment of a program. Like text and data symbols, the value of a bss symbol may change during a subsequent link-editor run, since previously loaded programs may have bss segments. After the first .bss statement, the value of ``.'' is bss 0.

external absolute, text, data, or bss

Symbols declared .globl but defined within an assembly as absolute, text, data, or bss symbols may be used exactly as if they were not declared .globl; however, their value and type are available to the link editor so that the program may be loaded with others that reference these symbols.

other types
: Each keyword known to the assembler has a type which is used to select the routine which processes the associated keyword statement. The behavior of such symbols when not used as keywords is the same as if they were absolute.

2.5.6.3 Type propagation in expressions When operands are combined by expression operators, the result has a type which depends on the types of the operands and on the operator. The rules involved are complex to state but were intended to be sensible and predictable. For purposes of expression evaluation the important types are

undefined
absolute
text
data
bss
undefined external
other

The combination rules are then: If one of the operands is undefined, the result is undefined. If both operands are absolute, the result is absolute. If an absolute is combined with one of the `other types'mentioned above, the result has the other type. If two operands of `other type' are combined, the result has the numerically larger type. An `other type' combined with an explicitly discussed type other than absolute acts like an absolute.

Further rules applying to particular operators are:

\+ If one operand is text-, data-, or bss-segment relocatable, or is an undefined external, the result has the postulated type and the other operand must be absolute.

\- If the first operand is a relocatable text-, data-, or bss-segment symbol, the second operand may be absolute (in which case the result has the type of the first operand); or the second operand may have the same type as the first (in which case the result is absolute). If the first operand is external undefined, the second must be absolute. All other combinations are illegal.

others
: It is illegal to apply these operators to any but absolute symbols.

2.5.7 Pseudo-operations

The keywords listed below introduce statements that influence the later operations of the assembler. The metanotation

[stuff] ...

means that 0 or more instances of the given stuff may appear. Also, boldface tokens are literals, italic words are substitutable.

2.5.7.1 .even If the location counter ``.'' is odd, it is advanced by one so the next statement will be assembled at a word boundary. This is useful for forcing storage allocation to be on a word boundary after a .byte or .ascii directive.

2.5.7.2 .float, .double

.float 31459E4

The .float psuedo operation accepts as its operand an optional string of tabs and spaces, then an optional sign, then a string of digits optionally containing a decimal point, them an optional `e' or `E', followed by an optionally signed integer. The string is interpreted as a floating point number. The difference between .float and .double is in the number of bytes for the result; .float sets aside four bytes, while .double sets aside eight bytes.

2.5.7.3 .B .globl

.globl *name* [, *name*] ...

This statement makes the *names* external. If they are otherwise defined (by assignment or appearance as a label) they act within the assembly exactly as if the .globl statement were not given; however, the link editor ld may be used to combine this routine with other routines that refer to these symbols.

Conversely, if the given symbols are not defined within the current assembly, the link editor can combine the output of this assembly with that of others which define the symbols. It is possible to force the assembler to make all otherwise undefined symbols external.

2.5.7.4 .text, .data, .bss These three pseudo-operations cause the assembler to begin assembling into the text, data, or bss segment respectively. Assembly starts in the text segment. It is forbidden to assemble any code or data into the bss segment, but symbols may be defined and ``.'' moved about by assignment.

2.5.7.5 .comm The format of the .comm is:

.comm ARRAY

Provided the name is not defined elsewhere, this statement is equivalent to .globl. That is, the type of name is ``undefined external'', and its size is expression. In fact the name behaves in the current assembly just like an undefined external. However, the link-editor ld has been special-cased so that all external symbols which are not otherwise defined, and which have a non-zero value, are defined to lie in the bss segment, and enough space is left after the symbol to hold expression bytes. All symbols which become defined in this way are located before all the explicitly defined bss-segment locations.

2.5.7.6 .insrt The format of a .insrt is:

.insrt "filename"

where filename is any valid XENIX filename. Note that the filename must be enclosed within double quotes.

The assembler will attempt to open this file for input. If it succeeds, source lines will be read from it until the end of file is reached. If as was unable to open the file, a Cannot open insert file error message will be generated.

This statement is useful for including a standard set of comments or symbol assignments at the beginning of a program. It is also useful for breaking up a large source program into easily managable pieces.

A maximum depth of 10 (ten) files may be .insrted at any one time.

System call names are not predefined. They may be found in the file /usr/include/sys.s.

2.5.7.7 .ascii, .asciz The .ascii directive translates character strings into their 7-bit ascii (represented as 8-bit bytes) equivalents for use in the source program. The format of the .ascii directive is as follows:

.ascii /character string/

where

character string contains any character valid in a character constant. Obviously, a <newline> must not appear within the character string. (It can be represented by the escape sequence \en).

/ and / are delimiter characters, which may be any character not appearing in character string

Several examples follow:

Hex Code Generated:	Statement:
22 68 65 6C 6C 6F 20 74	.ascii /"hello there"/
68 65 72 65 22	
77 61 72 6E 69 6E 67 20	.ascii "Warning-\007\007 \n"
2D 07 07 20 0A	
61 62 63 64 65 66 67	.ascii *abcdefg*

The .asciz directive is equivalent to the .ascii directive with a zero (null) byte automatically inserted as the final character of the string. Thus, when a list or text string is to be printed, a search for the null character can terminate the string. Null terminated strings are used as arguments to some XENIX system calls.

2.5.7.8 .list, .nlist These pseudo-directives control the assembler output listing. These, in effect, temporarily override the `-l' switch to the assembler. This is useful when certain portions of the assembly output is not necessarily desired on a printed listing.

.list turns the listing on
.nlist turns the listing off

2.5.7.9 .blkb, .blkw The .blkb and .blkw directives are used to reserve blocks of storage: .blkb reserves bytes, .blkw reserves words.

The format is:

```
.blkb       [expression]
.blkw       [expression]
```

where expression is the number of bytes or words to reserve. If no argument is given a value of 1 is assumed. The expression must be absolute, and defined during pass 1.

This is equivalent to the statement ``.=.+expression'', but has a much more transparent meaning.

2.5.7.10 .byte, .word The .byte and .word directives are used to reserve bytes and words and to initialize them with certain values.

The format is:

```
.byte       [expression]
.word       [expression]
```

The .byte directive reserves one byte for each expression in the operand field and initializes the value of the byte to be the low-order byte of the corresponding expression.

For example,

```
.byte 0
                reserves an byte, with a value
                of zero.
state: .byte 0
                reserves a byte with a zero
                value called state.
```

The semantics for .word are identical, except that 16-bit words are reserved and initialized.

2.5.7.11 .end The .end directive indicates the physical end of the source program. The format is:

```
.end        [expression]
```

where expression is an optional argument which, if present, indicates the entry point of the program, i.e. the starting point for execution. If the entry point of a program is not specified during assembly, it defaults to zero.

Every source program must be terminated with a .end statement. Inserted files which contain a .end statement will terminate assembly of the entire program, not just the inserted portion.

2.5.8 Machine

The 8086 instructions treat different types of operands uniformly. Nearly every instruction can operate on either byte or word data. In the table that follows, with some notable exceptions, an instruction that operates on a byte operand will have a b suffix on the opcode.

The 8086 instruction mnemonics which follow are implemented by the Microsoft 8086 assembler desribed in this document. Some of the opcodes are not found in any other 8086 manual.

For example, this document describes branch instructions not found in any 8086 manual. The branch instructions expand into a jump on the inverse of the condition specified, followed by an an unconditional intra-segment jump. The operand field format for the branch opcodes is the same as the operand field for the jump long opcodes. The opcodes which are implemented only in this assembler will be annotated by an asterisk, and will be fully defined and described in this document.

8086 Assembler Opcodes

Opcode	Description
aaa	ascii adjust for addition
aad	ascii adjust for division
aam	ascii adjust for multiply
aas	ascii adjust for subtraction
adc	add with carry
adcb	add with carry
add	add
addb	add
and	logical AND
andb	logical AND
*beq	long branch equal
*bge	long branch grt or equal
*bgt	long branch grt
*bhi	long branch on high
*bhis	long branch high or same
*ble	long branch les or equal
*blo	long branch on low
*blos	long branch low or same
*blt	long branch less than
*bne	long branch not equal
*br	long branch
call	intra segment call
calli	inter segment call
cbw	convert byte to word
clc	clear carry flag
cld	clear direction flag
cli	clear interrupt flag
cmc	complement carry flag
cmp	compare
cmpb	compare
cmps	compare string
cmpsb	compare string
cwd	covert word to double word
daa	decimal adjust for addition
das	decimal adjust for subtraction
dec	decrement by one
decb	decrement by one
div	divison unsigned
divb	divison unsigned
hlt	halt
idiv	integer division
idivb	integer division
imul	integer multiplication
imulb	integer multiplication
in	input byte
inc	increment by one
incb	increment by one
int	interrupt

into	interrupt if overflow
inw	input word
iret	interrupt return
j	short jump
ja	short jump if above
jae	short jump if above or equal
jb	short jump if below
jbe	short jump if below or equal
jcxz	short jump if CX is zero
je	short jump on equal
jg	short jump on greater than
jge	short jump greater than or equal
jl	short jump on less than
jle	short jump on less than or equal
jmp	jump
jmpi	inter segment jump
jna	short jump not above
jnae	short jump not above or equal
jnb	short jump not below
jnbe	short jump not below or equal
jne	short jump not equal
jng	short jump not greater
jnge	short jump not greater or equal
jnl	short jump not less
jnle	short jump not less or equal
jno	short jump not overflow
jnp	short jump not parity
jns	short jump not sign
jnz	short jump not zero
jo	short jump on overflow
jp	short jump if parity
jpe	short jump if parity even
jpo	short jump if parity odd
js	short jump if signed
jz	short jump if zero
lahf	load AH from flags
lds	load pointer using DS
lea	load effective address
les	load pointer using ES
lock	lock bus
lodb	load string byte
lodw	load string word
loop	loop short label
loope	loop if equal
loopne	loop if not equal
loopnz	loop is not zero
loopz	loop if zero
mov	move
movb	move byte
movs	move string
movsb	move string byte

mul	multipication unsigned
mulb	multipication unsigned
neg	negate
negb	negate
nop	no op
not	logical NOT
notb	logical NOT
or	Iogical OR
orb	logical OR
out	output byte
outw	output word
pop	pop from stack
popf	pop flag from stack
push	push onto stack
pushf	push flags onto stack
rcl	rotate left through carry
rclb	rotate left through carry
rcr	rotate right throuch carry
rcrb	rotate right throuch carry
rep	repeat string operation
repnz	repeat string operation not zero
repz	repeat string operation while zero
ret	return from procedure
reti	return from intersegment procedure
rol	rotate left
rolb	rotate left
ror	rotate right
rorb	rotate right
sahf	store AH into flagsno operands
sal	shift arithmetic left
salb	shift arithmetic left
sar	shift arithmetic right
sarb	shift arithmetic right
sbb	subtract with borrow
sbbb	subtract with borrow
scab	scan string
shl	shift logical left
shlb	shift logical left
shr	shidr logical right
shrb	shidr logical right
stc	set carry flag
std	set direction flag
sti	set interrupt enable flag
stob	store byte string
stow	store word string
sub	subtraction
subb	subtraction
test	test
testb	test
wait	wait while TEST pin
xchg	exchange

```
xchgb     exchange
xlat      translate
xor       xclusive OR
xorb      xclusive OR
```

2.5.9 Addressing Modes

The 8086 assembler provides many different ways to access instruction operands. Operands may be contained in registers, within the instruction itself, in memory, or in I/O ports. In addition, the addresses of memory and I/O port operands can be calculated in several different ways.

2.5.9.1 Register Operands Instructions that specify only register operands are generally the most compact and fastest executing of all the instruction forms. This is because the register `addresses' are encoded in the instructions with just a few bits, and because these operations are performed entirely within the CPU. Registers may serve as source operands, destination operands, or both.

EXAMPLES OF REGISTER ADDRESSING

```
sub       cx,di
mv        ax,/3*4
mv        /3*4/,ax
mov       ax,*1
```

2.5.9.2 Immediate Operands Immediate operands are constant data contained in an instruction. The data may be either 8 or 16 bits in length. Immediate operands can be accessed quickly because they are available directly from the instruction queue; it is possible that no bus cycles will be needed to obtain an immediate operand. An immediate operand is always a constant value and can only be used as a source operand.

The assembler can accept both 8 and 16 bit operands. It does not perform any checking on the operand size, but determines the size of the operand by the following symbols:

```
*expr     an 8 bit immediate
#expr     a 16 bit immediate
```

EXAMPLES OF IMMEDIATE ADDRESSING

```
mov     cx,*PAGSIZ/2
mov     cx,#PAGSIZ/2
mov     map,#PAGSIZ/2
mov     map,*PAGSIZ/2
```

2.5.10 Memory Addressing Modes

When reading or writing a memory operand, a value called the offset is required. This offset value, also called the effective address is the operand's distance in bytes from the beginning of the segment in which it resides.

2.5.10.1 Direct Addressing Direct addressing is the simplest memory addressing mode since no registers are involved. The effective address is taken directly from the displacement field of the instruction. It is typically used to access simple (scalar) variables.

EXAMPLES OF DIRECT ADDRESSING

```
push    *6(bp)
mov     cx,#256
add     si,*4
```

2.5.10.2 Register Indirect Addressing The effective address of a memory operand may be taken from a base or index register. One instruction can operate on many different memory locations if the value in the base or index register is updated appropriately. Indirect addressing is denoted by an ampersand @ preceding the operand.

EXAMPLES OF INDIRECT ADDRESSING

```
popl    rr0,@r15
calli   @moncall
```

2.5.10.3 Based Addressing In based addressing, the effective address is the sum of a displacement value and the content of register bx or bp. Based addressing also provides a straightforward way to address structures which may be located in different places in memory. A base register can be pointed at the base of the structure and elements of the structure addressed by their displacements from the base. Different copies of the same structure can be accessed by simply changing the base register.

EXAMPLE OF BASED ADDRESSING

```
mov         *2(si),#/1000
```

2.5.10.4 Indexed Addressing In indexed addressing, the effective address is calculated from the sum of a displacement plus the content of an index register. Indexed addressing often is used to access elements in an array. The displacement locates the beginnning of the array, and the value of the index register selects one element. Since all array elements are the same length, simple arithmetic on the index register will select any element.

EXAMPLE OF INDEXED ADDRESSING

```
mov         #_cat,(bx)
```

2.5.10.5 Based Indexed Addressing Based indexed addressing generates an effective address that is the sum of a base register, an index register, and a displacement. Based indexed addressing is a very flexible mode because two address components can be varied at execution time.

Based indexed addressing provides a convenient way for a procedure to address an array allocated on a stack. Register bp can contain the offset of a reference point on the stack, typically the top of the stack after the procedure has saved registers and allocated local storage. The offset of the beginning of the array from the reference point can be expressed by a displacement value, and an index register can be used to access individual array elements.

EXAMPLES OF BASED INDEXED ADDRESSING

```
mov         (bx)(dx),_sym
mov         *2(bx)(dx),_sym
mov         #2(bx)(dx),_sym
```

2.5.11 Diagnostics

When syntactic errors occur, the line number and the file in which they occur is displayed. Errors in pass 1 cause cancellation of pass 2.

```
***ERROR*** syntax error, line xx
file: vv errors
```

where xx represents the line number(s) in error, and yy represents the total number of errors.

Appendix I

TUTORIAL AND REFERENCE MATERIAL (UNIVERSITY OF CALIFORNIA, BERKELEY, BERKELEY MANUALS)

On the following pages is informational material developed at the University of California, Berkeley. The material is supplied under license from the Regents of the University.

An Introduction to the C Shell

An Introduction to Display Editing with Vi

Quick Reference for Ex, Vi

Ex Reference Manual

Edit: A Tutorial

Ex/Edit Command Summary

Mail Reference Manual

-ME Reference Manual

Screen Updating and Cursor Movement Optimization: A Library Package

An introduction to the C shell

(Revised for the Third Berkeley Distribution)

William Joy

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 shells 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.

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

December 20, 1979

†UNIX is a Trademark of Bell Laboratories.

An Introduction to Display Editing with Vi

William Joy

Revised for versions 3.5/2.13 by
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.

September 16, 1980

An Introduction to Display Editing with Vi

William Joy

Revised for versions 3.5/2.13 by
Mark Horton

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

1. Getting started

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

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

1.1. Specifying terminal type

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

Code	Full name	Type
2621	Hewlett-Packard 2621A/P	Intelligent
2645	Hewlett-Packard 264x	Intelligent
act4	Microterm ACT-IV	Dumb
act5	Microterm ACT-V	Dumb
adm3a	Lear Siegler ADM-3a	Dumb
adm31	Lear Siegler ADM-31	Intelligent
c100	Human Design Concept 100	Intelligent
dm1520	Datamedia 1520	Dumb
dm2500	Datamedia 2500	Intelligent
dm3025	Datamedia 3025	Intelligent
fox	Perkin-Elmer Fox	Dumb
h1500	Hazeltine 1500	Intelligent
h19	Heathkit h19	Intelligent
i100	Infoton 100	Intelligent
mime	Imitating a smart act4	Intelligent

The financial support of an IBM Graduate Fellowship and the National Science Foundation under grants MCS74-07644-A03 and MCS78-07291 is gratefully acknowledged.

t1061	Teleray 1061	Intelligent
vt52	Dec VT-52	Dumb

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

```
% setenv TERM 2621
```

This command works with the shell *csh* on both version 6 and 7 systems. If you are using the standard version 7 shell then you should give the commands

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

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

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

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

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

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

1.2. Editing a file

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

```
% vi name
```

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

1.3. 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.

---

‡ 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.

1.4. Notational conventions

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

1.5. Arrow keys

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

(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.

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

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

1.7. Getting out of the editor

After you have worked with this introduction for a while, and you wish to do something else, you can give the command **ZZ** to the editor. This will write the contents of the editor's buffer back into the file you are editing, if you made any changes, and then quit from the editor. You can also end an editor session by giving the command **:q!**CR;† this is a dangerous but occasionally essential command which ends the editor session and discards all your changes. You need to know about this command in case you change the editor's copy of a file you wish

---

• 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.

• Backspacing over the '/' will also cancel the search.

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

† All commands which read from the last display line can also be terminated with a ESC as well as an CR.

only to look at. Be very careful not to give this command when you really want to save the changes you have made.

2. Moving around in the file

2.1. Scrolling and paging

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

As you know now if you tried hitting ^D, this command scrolls down in the file. The D thus stands for down. Many editor commands are mnemonic and this makes them much easier to remember. For instance the command to scroll up is ^U. Many dumb terminals can't scroll up at all, in which case hitting ^U clears the screen and refreshes it with a line which is farther back in the file at the top.

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

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

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

2.2. Searching, goto, and previous context

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

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.*

---

‡ If you don't have a '^' key on your terminal then there is probably a key labelled '↑'; in any case these characters are one and the same.

‡‡ Version 3 only.

‡ Not available in all v2 editors due to memory constraints.

† 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 nowsCR.

*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.)

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. (On certain terminals using v2 editors, they won't.) 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.

2.4. Moving within a line

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

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

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

2.5. Summary

SPACE	advance the cursor one position
^B	backwards to previous page
^D	scrolls down in the file
^E	exposes another line at the bottom (v3)
^F	forward to next page
^G	tell what is going on
^H	backspace the cursor
^N	next line, same column
^P	previous line, same column
^U	scrolls up in the file
^Y	exposes another line at the top (v3)
+	next line, at the beginning
−	previous line, at the beginning
/	scan for a following string forwards
?	scan backwards
B	back a word, ignoring punctuation
G	go to specified line, last default
H	home screen line
M	middle screen line
L	last screen line
W	forward a word, ignoring punctuation
b	back a word
e	end of current word
n	scan for next instance of / or ? pattern
w	word after this word

2.6. View ‡

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

3. Making simple changes

3.1. Inserting

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

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

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

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

‡ Not available in all v2 editors due to memory constraints.

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 when you hit ESC; if you want to get rid of them immediately, hit an ESC and then a again.

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

3.2. Making small corrections

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

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

3.3. More corrections: operators

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

---

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

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

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

3.4. Operating on lines

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

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

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

3.5. Undoing

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

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

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

3.6. Summary

SPACE	advance the cursor one position
^H	backspace the cursor
^W	erase a word during an insert
erase	your erase (usually ^H or #), erases a character during an insert
kill	your kill (usually @, ^X, or ^U), kills the insert on this line
.	repeats the changing command
O	opens and inputs new lines, above the current
U	undoes the changes you made to the current line
a	appends text after the cursor
c	changes the object you specify to the following text

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

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

d	deletes the object you specify
i	inserts text before the cursor
o	opens and inputs new lines, below the current
u	undoes the last change

4. Moving about; rearranging and duplicating text

4.1. Low level character motions

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

When you are operating on the text in a line it is often desirable to deal with the characters up to, but not including, the first instance of a character. Try d*fx* for some *x* now and notice that the *x* character is deleted. Undo this with **u** and then try d*tx*; 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.

4.2. Higher level text objects

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

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

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

---

* This is settable by a command of the form :se ts=x<CR>, where *x* is 4 to set tabstops every four columns. This has effect on the screen representation within the editor.

† The [[and]] operations require the operation character to be doubled because they can move the cursor far

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

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

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

4.3. Rearranging and duplicating text

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

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

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

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

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

---

from where it currently is. While it is easy to get back with the command ', these commands would still be frustrating if they were easy to hit accidentally.

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

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 *x* forward in line
p	put text back, after cursor or below current line
y	yank operator, for copies and moves
tx	up to *x* 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 :!*cmd*CR. The system will run the single command *cmd* and when the command finishes, the editor will ask you to hit a RETURN to continue. When you have finished looking at the output on the screen, you should hit RETURN and the editor will clear the screen and redraw it. You can then continue editing. You can also give another : command when it asks you for a RETURN; in this case the screen will not be redrawn.

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

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

5.3. Marking and returning

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

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

5.4. Adjusting the screen

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

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

Finally, if you wish to place a certain line on the screen at the top middle or bottom of the screen, you can position the cursor to that line, and then give a **z** command. You should follow the **z** command with a RETURN if you want the line to appear at the top of the window, a **.** if you want it at the center, or a **-** if you want it at the bottom. (**z.**, **z-**, and **z+** are not available on all v2 editors.)

6. Special topics

6.1. Editing on slow terminals

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

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

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

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

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

: / ? [[ ]] ` '

Thus if you are searching for a particular instance of a common string in a file you can precede

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

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

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 no*opt*

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

You can get a list of all options which you have changed by the command :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.

Options set by the set command only last while you stay in the editor. It is common to want to have certain options set whenever you use the editor. This can be accomplished by creating a list of *ex* commands† which are to be run every time you start up *ex*, *edit*, or *vi*. A

† Note that the command 5z. has an entirely different effect, placing line 5 in the center of a new window.
† All commands which start with : are *ex* commands.

typical list includes a **set** command, and possibly a few **map** commands (on v3 editors). Since it is advisable to get these commands on one line, they can be separated with the | character, for example:

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

which sets the options *autoindent*, *autowrite*, *terse*, (the **set** command), makes @ delete a line, (the first **map**), and makes # delete a character, (the second **map**). (See section 6.9 for a description of the **map** command, which only works in version 3.) This string should be placed in the variable EXINIT in your environment. If you use *csh*, put this line in the file *.login* in your home directory:

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

If you use the standard v7 shell, put these lines in the file *.profile* in your home directory:

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

On a version 6 system, the concept of environments is not present. In this case, put the line in the file *.exrc* in your home directory.

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

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

6.3. Recovering lost lines

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

"1pu.u.u.

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

6.4. Recovering lost files

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

% vi −r *name*

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

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

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

```
% vi -r
```

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

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

* This feature is not available on some v2 editors. In v2 editors where it is available, the break can only occur to the right of the specified boundary instead of to the left.

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 a comment quickly.

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

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

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

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

6.9. Macros‡

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

Briefly, there are two flavors of macros:

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

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

:map *lhs rhs*CR

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

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

:map q :wq^V^VCR CR

which means that whenever you type q, it will be as though you had typed the four characters :wqCR. A ^V's is needed because without it the CR would end the : command, rather than

† The LISP features are not available on some v2 editors due to memory constraints.

‡ The macro feature is available only in version 3 editors.

becoming part of the *map* definition. There are two ^V's because from within *vi*, two ^V's must be typed to get one. The first CR is part of the *rhs*, the second terminates the : command.

Macros can be deleted with

unmap lhs

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

:map ^V^V^I #

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

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

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

:map ^T ^V␣␣␣␣

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

7. Word Abbreviations ‡‡

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

:ab eecs Electrical Engineering and Computer Sciences

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

7.1. Abbreviations

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

8. Nitty-gritty details

8.1. Line representation in the display

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

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

‡‡ Version 3 only.

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

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:

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

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

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

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

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

8.3. More file manipulation commands

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

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

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

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. (The tag feature is not available in some v2 editors.)

8.4. More about searching for strings

When you are searching for strings in the file with / and ?, the editor normally places you at the next or previous occurrence of the string. If you are using an operator such as d, c or y, then you may well wish to affect lines up to the line before the line containing the pattern.

You can give a search of the form */pat/−n* to refer to the *n*'th line before the next line containing *pat*, or you can use + instead of − to refer to the lines after the one containing *pat*. If you don't give a line offset, then the editor will affect characters up to the match place, rather than whole lines; thus use "+0" to affect to the line which matches.

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

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

set nomagic

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

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

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

8.5. More about input mode

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

^H	deletes the last input character
^W	deletes the last input word, defined as by b
erase	your erase character, same as ^H
kill	your kill character, deletes the input on this line
\	escapes a following ^H and your erase and kill
ESC	ends an insertion
DEL	interrupts an insertion, terminating it abnormally
CR	starts a new line
^D	backtabs over *autoindent*
0^D	kills all the *autoindent*
↑^D	same as 0^D, but restores indent next line
^V	quotes the next non-printing character into the file

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

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

The command A which appends at the end of the current line is often useful for continuing.

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

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

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

---

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

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

‡ Not available in all v2 editors due to memory constraints.

displayed.

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

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

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

Acknowledgements

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

Appendix: character functions

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

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

^@ Not a command character. If typed as the first character of an insertion it is replaced with the last text inserted, and the insert terminates. Only 128 characters are saved from the last insert; if more characters were inserted the mechanism is not available. A ^@ cannot be part of the file due to the editor implementation (7.5f).

^A Unused.

^B Backward window. A count specifies repetition. Two lines of continuity are kept if possible (2.1, 6.1, 7.2).

^C Unused.

^D As a command, scrolls down a half-window of text. A count gives the number of (logical) lines to scroll, and is remembered for future ^D and ^U commands (2.1, 7.2). During an insert, backtabs over *autoindent* white space at the beginning of a line (6.6, 7.5); this white space cannot be backspaced over.

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

^F Forward window. A count specifies repetition. Two lines of continuity are kept if possible (2.1, 6.1, 7.2).

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

^H (BS) Same as **left arrow**. (See **h**). During an insert, eliminates the last input character, backing over it but not erasing it; it remains so you can see what you typed if you wish to type something only slightly different (3.1, 7.5).

^I (TAB) Not a command character. When inserted it prints as some number of spaces. When the cursor is at a tab character it rests at the last of the spaces which represent the tab. The spacing of tabstops is controlled by the *tabstop* option (4.1, 6.6).

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

^K Unused.

^L The ASCII formfeed character, this causes the screen to be cleared and redrawn. This is useful after a transmission error, if characters typed by a program other than the editor scramble the screen, or after output is stopped by an interrupt (5.4, 7.2f).

^M (CR) A carriage return advances to the next line, at the first non-white position in the line. Given a count, it advances that many lines (2.3). During an insert, a CR causes the insert to continue onto another line (3.1).

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

^O Unused.

^P — Same as up arrow (see k).

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

^R — Redraws the current screen, eliminating logical lines not corresponding to physical lines (lines with only a single @ character on them). On hardcopy terminals in *open* mode, retypes the current line (5.4, 7.2, 7.8).

^S — Unused. Some teletype drivers use ^S to suspend output until ^Qis

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

^U — Scrolls the screen up, inverting ^D which scrolls down. Counts work as they do for ^D, and the previous scroll amount is common to both. On a dumb terminal, ^U will often necessitate clearing and redrawing the screen further back in the file (2.1, 7.2).

^V — Not a command character. In input mode, quotes the next character so that it is possible to insert non-printing and special characters into the file (4.2, 7.5).

^W — Not a command character. During an insert, backs up as b would in command mode; the deleted characters remain on the display (see ^H) (7.5).

^X — Unused.

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

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

^[(ESC) — Cancels a partially formed command, such as a z when no following character has yet been given; terminates inputs on the last line (read by commands such as : / and ?); ends insertions of new text into the buffer. If an ESC is given when quiescent in command state, the editor rings the bell or flashes the screen. You can thus hit ESC if you don't know what is happening till the editor rings the bell. If you don't know if you are in insert mode you can type ESCa, and then material to be input; the material will be inserted correctly whether or not you were in insert mode when you started (1.5, 3.1, 7.5).

^\ — Unused.

^] — Searches for the word which is after the cursor as a tag. Equivalent to typing :ta, this word, and then a CR. Mnemonically, this command is "go right to" (7.3).

^^ — Equivalent to :e #CR, returning to the previous position in the last edited file, or editing a file which you specified if you got a 'No write since last change diagnostic' and do not want to have to type the file name again (7.3). (You have to do a :w before ^^ will work in this case. If you do not wish to write the file you should do :e! #CR instead.)

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

SPACE — Same as right arrow (see l).

! — An operator, which processes lines from the buffer with reformatting commands. Follow ! with the object to be processed, and then the command name terminated by CR. Doubling ! and preceding it by a count causes count lines to be filtered; otherwise the count is passed on to the object after the !. Thus 2!}*fmt*CR reformats the next two paragraphs by running them through the program *fmt*. If you are working on LISP, the command !%*grind*CR,* given at the

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

beginning of a function, will run the text of the function through the LISP grinder (6.7, 7.3). To read a file or the output of a command into the buffer use :r (7.3). To simply execute a command use :! (7.3).

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

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

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

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

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

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

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

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

* Unused.

+ Same as CR when used as a command.

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

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

. Repeats the last command which changed the buffer. Especially useful when deleting words or lines; you can delete some words/lines and then hit . to delete more and more words/lines. Given a count, it passes it on to the command being repeated. Thus after a 2dw, 3. deletes three words (3.3, 6.3, 7.2, 7.4).

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

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

To include the character / in the search string, you must escape it with a preceding \. A ↑ at the beginning of the pattern forces the match to occur at the beginning of a line only; this speeds the search. A $ at the end of the pattern forces the match to occur at the end of a line only. More extended pattern matching is available, see section 7.4; unless you set **nomagic** in your *.exrc* file you will have to preceed the characters . [* and ~ in the search pattern with a \ to get them to work as you would naively expect (1.5, 2.2, 6.1, 7.2, 7.4).

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

1−9 Used to form numeric arguments to commands (2.3, 7.2).

: A prefix to a set of commands for file and option manipulation and escapes to the system. Input is given on the bottom line and terminated with an CR, and the command then executed. You can return to where you were by hitting DEL or RUB if you hit : accidentally (see primarily 6.2 and 7.3).

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

< An operator which shifts lines left one *shiftwidth*, normally 8 spaces. Like all operators, affects lines when repeated, as in <<. Counts are passed through to the basic object, thus 3<< shifts three lines (6.6, 7.2).

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

> An operator which shifts lines right one *shiftwidth*, normally 8 spaces. Affects lines when repeated as in >>. Counts repeat the basic object (6.6, 7.2).

? Scans backwards, the opposite of /. See the / description above for details on scanning (2.2, 6.1, 7.4).

@ A macro character (6.9). If this is your kill character, you must escape it with a \ to type it in during input mode, as it normally backs over the input you have given on the current line (3.1, 3.4, 7.5).

A Appends at the end of line, a synonym for $a (7.2).

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

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

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

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

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

G Goes to the line number given as preceding argument, or the end of the file if no preceding count is given. The screen is redrawn with the new current line in the center if necessary (7.2).

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

I Inserts at the beginning of a line; a synonym for ↑I.

J Joins together lines, supplying appropriate white space: one space between words, two spaces after a ., and no spaces at all if the first character of the joined on line is). A count causes that many lines to be joined rather than the default two (6.5, 7.1f).

K Unused.

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

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

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

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

P Puts the last deleted text back before/above the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines. Otherwise the text is inserted between the characters before and at the cursor. May be preceded by a named buffer specification "*x* to retrieve the contents of the buffer; buffers **1**−**9** contain deleted material, buffers **a**−**z** are available for general use (6.3).

Q Quits from *vi* to *ex* command mode. In this mode, whole lines form commands, ending with a RETURN. You can give all the : commands; the editor supplies the : as a prompt (7.7).

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

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

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

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

V Unused.

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

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

Y Yanks a copy of the current line into the unnamed buffer, to be put back by a later **p** or **P**; a very useful synonym for **yy**. A count yanks that many lines. May be preceded by a buffer name to put lines in that buffer (7.4).

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

[[Backs up to the previous section boundary. A section begins at each macro in the *sections* option, normally a '.NH' or '.SH' and also at lines which which start with a formfeed ^L. Lines beginning with { also stop **[[**; this makes it useful for looking backwards, a function at a time, in C programs. If the option *lisp* is set, stops at each (at the beginning of a line, and is thus useful for moving backwards at the top level LISP objects. (4.2, 6.1, 6.6, 7.2).

**** Unused.

]] Forward to a section boundary, see **[[** for a definition (4.2, 6.1, 6.6, 7.2).

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

_ Unused.

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

a Appends arbitrary text after the current cursor position; the insert can continue onto multiple lines by using RETURN within the insert. A count causes the inserted text to be replicated, but only if the inserted text is all on one line. The insertion terminates with an ESC (3.1, 7.2).

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

c An operator which changes the following object, replacing it with the following input text up to an ESC. If more than part of a single line is affected, the text which is changed away is saved in the numeric named buffers. If only part of the current line is affected, then the last character to be changed away is marked with a **$**. A count causes that many objects to be affected, thus both **3e)** and **c3)** change the following three sentences (7.4).

d An operator which deletes the following object. If more than part of a line is affected, the text is saved in the numeric buffers. A count causes that many objects to be affected; thus **3dw** is the same as **d3w** (3.3, 3.4, 4.1, 7.4).

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

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

g Unused.

Arrow keys **h**, **j**, **k**, **l**, and **H**.

h **Left arrow**. Moves the cursor one character to the left. Like the other arrow keys, either **h**, the **left arrow** key, or one of the synonyms (^H) has the same effect. On v2 editors, arrow keys on certain kinds of terminals (those which send escape sequences, such as vt52, c100, or hp) cannot be used. A count repeats the effect (3.1, 7.5).

i Inserts text before the cursor, otherwise like **a** (7.2).

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

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

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

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

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

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

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

q Unused.

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

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

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

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

v Unused.

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

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

y An operator, yanks the following object into the unnamed temporary buffer. If preceded by a named buffer specification, "x, the text is placed in that buffer also. Text can be recovered by a later **p** or P (7.4).

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

{	Retreats to the beginning of the beginning of the preceding paragraph. A paragraph begins at each macro in the *paragraphs* option, normally '.IP', '.LP', '.PP', '.QP' and '.bp'. A paragraph also begins after a completely empty line, and at each section boundary (see [[above) (4.2, 6.8, 7.6).
|	Places the cursor on the character in the column specified by the count (7.1, 7.2).
}	Advances to the beginning of the next paragraph. See { for the definition of paragraph (4.2, 6.8, 7.6).
~	Unused.
^? (DEL)	Interrupts the editor, returning it to command accepting state (1.5, 7.5)

Ex Quick Reference

Entering/leaving ex

% ex *name*	edit *name*, start at end
% ex +*n name*	... at line *n*
% ex −t *tag*	start at *tag*
% ex −r	list saved files
% ex −r *name*	recover file *name*
% ex *name* ...	edit first; rest via :n
% ex −R *name*	read only mode
: x	exit, saving changes
: q!	exit, discarding changes

Ex states

Command	Normal and initial state. Input prompted for by :. Your kill character cancels partial command.
Insert	Entered by a i and c. Arbitrary text then terminates with line having only . character on it or abnormally with interrupt.
Open/visual	Entered by open or vi, terminates with Q or ^\.

Ex commands

abbrev	ab	next	n	unabbrev	una
append	a	number	nu	undo	u
args	ar	open	o	unmap	unm
change	c	preserve	pre	version	ve
copy	co	print	p	visual	vi
delete	d	put	pu	write	w
edit	e	quit	q	xit	x
file	f	read	re	yank	ya
global	g	recover	rec	*window*	z
insert	i	rewind	rew	*escape*	!
join	j	set	se	*lshift*	<
list	l	shell	sh	*print next*	CR
map		source	so	*resubst*	&
mark	ma	stop	st	*rshift*	>
move	m	substitute	s	*scroll*	^D

Ex command addresses

n	line *n*	/*pat*	next with *pat*
.	current	?*pat*	previous with *pat*
$	last	*x*-*n*	*n* before *x*
+	next	*x*,*y*	*x* through *y*
−	previous	'x	marked with x
+*n*	*n* forward	''	previous context
%	1,$		

Specifying terminal type

% setenv TERM *type*	*csh* and all version 6
$ TERM=*type*; export TERM	*sh* in Version 7

See also *tset*(1)

Some terminal types

2621	43	adm31	dw1	h19
2645	733	adm3a	dw2	i100
300s	745	c100	gt40	mime
33	act4	dm1520	gt42	owl
37	act5	dm2500	h1500	t1061
4014	adm3	dm3025	h1510	vt52

Initializing options

EXINIT	place set's here in environment var.
set *x*	enable option
set no*x*	disable option
set *x*=*val*	give value *val*
set	show changed options
set *x*?	show value of option *x*

Useful options

autoindent	ai	supply indent
autowrite	aw	write before changing files
ignorecase	ic	in scanning
lisp		( ) { } are s-exp's
list		print ^I for tab, $ at end
magic		. [* special in patterns
number	nu	number lines
paragraphs	para	macro names which start ...
redraw		simulate smart terminal
scroll		command mode lines
sections	sect	macro names ...
shiftwidth	sw	for < >, and input ^D
showmatch	sm	to) and] as typed
slowopen	slow	choke updates during insert
window		visual mode lines
wrapscan	ws	around end of buffer?
wrapmargin	wm	automatic line splitting

Scanning pattern formation

^	beginning of line
$	end of line
.	any character
\<	beginning of word
\>	end of word
[*str*]	any char in *str*
[^*str*]	... not in *str*
[*x*−*y*]	... between *x* and *y*
*	any number of preceding

Vi Quick Reference

Entering/leaving vi

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

The display

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

Vi states

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

Counts before vi commands

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

Simple commands

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

Interrupting, cancelling

ESC	end insert or incomplete cmd
^?	(delete or rubout) interrupts
^L	reprint screen if ^? scrambles it

File manipulation

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

Positioning within file

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

Adjusting the screen

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

Marking and returning

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

Line positioning

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

Character positioning

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

Words, sentences, paragraphs

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

Commands for LISP

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

Corrections during insert

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

Insert and replace

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

Operators (double to affect lines)

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

Miscellaneous operations

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

Yank and put

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

Undo, redo, retrieve

u	undo last change
U	restore current line

Ex Reference Manual
Version 3.5/2.13 — September, 1980

William Joy

Revised for versions 3.5/2.13 by
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 *Exledit Command Summary,* and a *Vi Quick Reference* card.

September 16, 1980

Ex Reference Manual
Version 3.5/2.13 — September, 1980

William Joy

Revised for versions 3.5/2.13 by
Mark Horton

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

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. It 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**] [**−w***n*] [**−x**] [**−R**] [**+***command*] name ...

The most common case edits a single file with no options, i.e.:

ex name

The **−** command line option 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 editor or system crash, retrieving the last saved version of the named file or, if no file is specified, typing a list of saved files. The **−l** option sets up for editing LISP, setting the *showmatch* and *lisp* options. The **−w** option sets the default window size to *n*, and is useful on dialups to start in small windows. The **−x** option causes *ex* to prompt for a *key*, which is used to encrypt and decrypt the contents of the file, which should already be encrypted using the same key, see *crypt*(1). The **−R** option sets the *readonly* option at the start. ‡ *Name* arguments indicate files to be edited. An argument of the form **+***command* indicates that the editor should begin by

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.

‡ Not available in all v2 editors due to memory constraints.

executing the specified command. If *command* is omitted, then it defaults to "$", positioning the editor at the last line of the first file initially. Other useful commands here are scanning patterns of the form "/pat" or line numbers, e.g. "+100" starting at line 100.

2. File manipulation

2.1. Current file

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

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

2.2. Alternate file

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

2.3. Filename expansion

Filenames within the editor may be specified using the normal shell expansion conventions. In addition, the character '%' in filenames is replaced by the *current* file name and the character '#' by the *alternate* file name.†

2.4. Multiple files and named buffers

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

For saving blocks of text while editing, and especially when editing more than one file, *ex* has a group of named buffers. These are similar to the normal buffer, except that only a limited number of operations are available on them. The buffers have names *a* through *z*.‡

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 accidently overwriting the file. Read only mode is on when the *readonly* option is set. It can be turned on with the −R command line option, by the *view* command line invocation, or by setting the *readonly* option. It can be cleared by setting *noreadonly*. It is possible to write, even while in read only mode, by indicating that you really

---

* The *file* command will say "[Not edited]" if the current file is not considered edited.

† This makes it easy to deal alternately with two files and eliminates the need for retyping the name supplied on an *edit* command after a *No write since last change* diagnostic is received.

‡ It is also possible to refer to *A* through *Z*; the upper case buffers are the same as the lower but commands append to named buffers rather than replacing if upper case names are used.

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.*

5. Command structure

Most command names are English words, and initial prefixes of the words are acceptable abbreviations. The ambiguity of abbreviations is resolved in favor of the more commonly used commands.*

---

* As an example, the command *substitute* can be abbreviated 's' while the shortest available abbreviation for the *set* command is 'se'.

5.1. Command parameters

Most commands accept prefix addresses specifying the lines in the file upon which they are to have effect. The forms of these addresses will be discussed below. A number of commands also may take a trailing *count* specifying the number of lines to be involved in the command.† Thus the command "10p" will print the tenth line in the buffer while "delete 5" will delete five lines from the buffer, starting with the current line.

Some commands take other information or parameters, this information always being given after the command name.‡

5.2. Command variants

A number of commands have two distinct variants. The variant form of the command is invoked by placing an '!' immediately after the command name. Some of the default variants may be controlled by options; in this case, the '!' serves to toggle the default.

5.3. Flags after commands

The characters '#', 'p' and 'l' may be placed after many commands.** In this case, the command abbreviated by these characters is executed after the command completes. Since *ex* normally prints the new current line after each change, 'p' is rarely necessary. Any number of '+' or '−' characters may also be given with these flags. If they appear, the specified offset is applied to the current line value before the printing command is executed.

5.4. Comments

It is possible to give editor commands which are ignored. This is useful when making complex editor scripts for which comments are desired. The comment character is the double quote: ". Any command line beginning with " is ignored. Comments beginning with " may also be placed at the ends of commands, except in cases where they could be confused as part of text (shell escapes and the substitute and map commands).

5.5. Multiple commands per line

More than one command may be placed on a line by separating each pair of commands by a '|' character. However the *global* commands, comments, and the shell escape '!' must be the last command on a line, as they are not terminated by a '|'.

5.6. Reporting large changes

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

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.

---

† Counts are rounded down if necessary.

‡ Examples would be option names in a *set* command i.e. "set number", a file name in an *edit* command, a regular expression in a *substitute* command, or a target address for a *copy* command, i.e. "1,5 copy 25".

** A 'p' or 'l' must be preceded by a blank or tab except in the single special case 'dp'.

n	The *n*th line in the editor's buffer, lines being numbered sequentially from 1.
$	The last line in the buffer.
%	An abbreviation for "1,$", the entire buffer.
+n −n	An offset relative to the current buffer line.†
/pat/ ?pat?	Scan forward and backward respectively for a line containing *pat*, 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 *pat*, then the trailing / or ? may be omitted. If *pat* 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 *mark* command, using single lower case letters *x* 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*.

(.) append — abbr: **a**
text
.

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

---

† 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.

a!
text
.

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

args

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

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

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

c!
text
.

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

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

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

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

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

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

Used to begin an editing session on a new file. The editor first checks to see if the buffer has been modified since the last *write* command was issued. If it has been, a warning is issued and the command is aborted. The command otherwise deletes the entire contents of the editor buffer, makes the named file the current file and prints the new filename. After insuring that this file is sensible† the editor reads the file into its buffer.

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

† I.e., that it is not a binary file such as a directory, a block or character special file other than */dev/tty*, a terminal, or a binary or executable file (as indicated by the first word).

‡ If executed from within *open* or *visual*, the current line is initially the first line of the file.

e! *file*

The variant form suppresses the complaint about modifications having been made and not written from the editor buffer, thus discarding all changes which have been made before editing the new file.

e *+n file*

Causes the editor to begin at line *n* rather than at the last line; *n* may also be an editor command containing no spaces, 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.

---

* 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.

i!
text
.

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

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

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

j!

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

(.) **k** *x*

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

(. , .) **list** *count flags*

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

map *lhs rhs*

The *map* command is used to define macros for use in *visual* mode. *Lhs* should be a single character, or the sequence "#n", for n a digit, referring to function key *n*. When this character or function key is typed in *visual* mode, it will be as though the corresponding *rhs* had been typed. On terminals without function keys, you can type "#n". See section 6.9 of the "Introduction to Display Editing with Vi" for more details.

(.) **mark** *x*

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

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

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

next abbr: **n**

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

n!

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

n *filelist*
n **+***command filelist*

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

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

Prints each specified line preceded by its buffer line number. The current line is left at the last line printed.

(.) **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.

---

‡ Not available in all v2 editors due to memory constraints.
* 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.

Address '0' is legal for this command and causes the file to be read at the beginning of the buffer. Statistics are given as for the *edit* command when the *read* successfully terminates. After a *read* the current line is the last line read.‡

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

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

recover *file*

Recovers *file* from the system save area. Used after a accidental hangup of the phone** or a system crash** or *preserve* command. Except when you use *preserve* you will be notified by mail when a file is saved.

rewind abbr: **rew**

The argument list is rewound, and the first file in the list is edited.

rew!

Rewinds the argument list discarding any changes made to the current buffer.

set *parameter*

With no arguments, prints those options whose values have been changed from their defaults; with parameter *all* it prints all of the option values.

Giving an option name followed by a '?' causes the current value of that option to be printed. The '?' is unnecessary unless the option is Boolean valued. Boolean options are given values either by the form 'set *option*' to turn them on or 'set no*option*' to turn them off; string and numeric options are assigned via the form 'set *option*=value'.

More than one parameter may be given to *set*; they are interpreted left-to-right.

shell abbr: **sh**

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

source *file* abbr: **so**

Reads and executes commands from the specified file. *Source* commands may be nested.

(. , .) **substitute** */pat/repl/ options count flags* abbr: **s**

On each specified line, the first instance of pattern *pat* is replaced by replacement pattern *repl.* If the *global* indicator option character 'g' appears, then all instances are substituted; if the *confirm* indication character 'c' appears, then before each substitution the line to be substituted is typed with the string to be substituted marked with '↑' characters. By typing an 'y' one can cause the substitution to be performed, any other input causes no change to take place. After a *substitute* the current line is the last line substituted.

Lines may be split by substituting new-line characters into them. The newline in *repl* must be escaped by preceding it with a '\'. Other metacharacters available in *pat* and *repl* are described below.

---

‡ Within *open* and *visual* the current line is set to the first line read rather than the last.

** The system saves a copy of the file you were editing only if you have made changes to the file.

stop

Suspends the editor, returning control to the top level shell. If *autowrite* is set and there are unsaved changes, a write is done first unless the form **stop!** is used. This commands is only available where supported by the teletype driver and operating system.

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

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

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

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

ta *tag*

The focus of editing switches to the location of *tag*, switching to a different line in the current file where it is defined, or if necessary to another file.‡

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 it's 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.

‡ Not available in all v2 editors due to memory constraints.

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

Enters visual mode at the specified line. *Type* is optional and may be '—' , '↑' or '.' as in the *z* command to specify the placement of the specified line on the screen. By default, if *type* is omitted, the specified line is placed as the first on the screen. A *count* specifies an initial window size; the default is the value of the option *window*. See the document *An Introduction to Display Editing with Vi* for more details. To exit this mode, type Q.

visual file
visual +*n* file

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

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

Writes changes made back to *file*, printing the number of lines and characters written. Normally *file* is omitted and the text goes back where it came from. If a *file* is specified, then text will be written to that file.* If the file does not exist it is created. The current file name is changed only if there is no current file name; the current line is never changed.

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

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

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

w! *name*

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

(1 , $) **w** !*command*

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

wq *name*

Like a *write* and then a *quit* command.

wq! *name*

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

xit *name*

If any changes have been made and not written, writes the buffer out. Then, in any case, quits.

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

Places the specified lines in the named *buffer*, for later retrieval via *put*. If no buffer name is specified, the lines go to a more volatile place; see the *put* command description.

---

* The editor writes to a file only if it is the current file and is *edited*, if the file does not exist, or if the file is actually a teletype, */dev/tty*, */dev/null*. Otherwise, you must give the variant form **w!** to force the write.

(.+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.

---

* 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.

(. , .) & *options count flags*

Repeats the previous *substitute* command.

(. , .) ~ *options count flags*

Replaces the previous regular expression with the previous replacement pattern from a substitution.

8. Regular expressions and substitute replacement patterns

8.1. Regular expressions

A regular expression specifies a set of strings of characters. A member of this set of strings is said to be *matched* by the regular expression. *Ex* remembers two previous regular expressions: the previous regular expression used in a *substitute* command and the previous regular expression used elsewhere (referred to as the previous *scanning* regular expression.) The previous regular expression can always be referred to by a null *re*, e.g. '//' or '??'.

8.2. Magic and nomagic

The regular expressions allowed by *ex* are constructed in one of two ways depending on the setting of the *magic* option. The *ex* and *vi* default setting of *magic* gives quick access to a powerful set of regular expression metacharacters. The disadvantage of *magic* is that the user must remember that these metacharacters are *magic* and precede them with the character '\' to use them as "ordinary" characters. With *nomagic*, the default for *edit*, regular expressions are much simpler, there being only two metacharacters. The power of the other metacharacters is still available by preceding the (now) ordinary character with a '\'. Note that '\' is thus always a metacharacter.

The remainder of the discussion of regular expressions assumes that that the setting of this option is *magic*.†

8.3. Basic regular expression summary

The following basic constructs are used to construct *magic* mode regular expressions.

char	An ordinary character matches itself. The characters '↑' at the beginning of a line, '$' at the end of line, '.' as any character other than the first, '.', '\', '[', and '~' are not ordinary characters and must be escaped (preceded) by '\' to be treated as such.
↑	At the beginning of a pattern forces the match to succeed only at the beginning of a line.
$	At the end of a regular expression forces the match to succeed only at the end of the line.
.	Matches any single character except the new-line character.
\<	Forces the match to occur only at the beginning of a "variable" or "word"; that is, either at the beginning of a line, or just before a letter, digit, or underline and after a character not one of these.
\>	Similar to '\<', but matching the end of a "variable" or "word", i.e. either the end of the line or before character which is neither a letter, nor a digit, nor the underline character.

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

[string] Matches any (single) character in the class defined by *string*. Most characters in *string* define themselves. A pair of characters separated by '−' in *string* defines the set of characters collating between the specified lower and upper bounds, thus '[a−z]' as a regular expression matches any (single) lower-case letter. If the first character of *string* is 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.

† When nested, parenthesized subexpressions are present, *n* is determined by counting occurrences of '\(' starting from the left.

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 repositions at the beginning but without retaining the previous indent.

Autoindent doesn't happen in *global* commands or when the input is not a terminal.

autoprint, ap default: ap

Causes the current line to be printed after each *delete*, *copy*, *join*, *move*, *substitute*, *t*, *undo* or shift command. This has the same effect as supplying a trailing 'p' to each such command. *Autoprint* is suppressed in globals, and only applies to the last of many commands on a line.

autowrite, aw default: noaw

Causes the contents of the buffer to be written to the current file if you have modified it and give a *next*, *rewind*, *stop*, *tag*, or *!* command, or a ^↑ (switch files) or ^] (tag goto) command in *visual*. Note, that the *edit* and *ex* commands do **not** autowrite. In each case, there is an equivalent way of switching when autowrite is set to avoid the *autowrite* (*edit* for *next*, *rewind!* for .I rewind , *stop!* for *stop*, *tag!* for *tag*, *shell* for *!*, and :e # and a :ta! command from within *visual).*

beautify, bf default: nobeautify

Causes all control characters except tab, newline and form-feed to be discarded from the input. A complaint is registered the first time a backspace character is discarded. *Beautify* does not apply to command input.

directory, dir default: dir=/tmp

Specifies the directory in which *ex* places its buffer file. If this directory in not writable, then the editor will exit abruptly when it fails to be able to create its buffer there.

edcompatible default: noedcompatible

Causes the presence of absence of g and c suffixes on substitute commands to be remembered, and to be toggled by repeating the suffices. The suffix r makes the substitution be as in the ~ command, instead of like &. ‡‡

errorbells, eb default: noeb

Error messages are preceded by a bell.* If possible the editor always places the error message in a standout mode of the terminal (such as inverse video) instead of ringing the bell.

hardtabs, ht default: ht=8

Gives the boundaries on which terminal hardware tabs are set (or on which the system expands tabs).

ignorecase, ic default: noic

‡‡ Version 3 only.

* Bell ringing in *open* and *vsual* on errors is not suppressed by setting *noeb*.

All upper case characters in the text are mapped to lower case in regular expression matching. In addition, all upper case characters in regular expressions are mapped to lower case except in character class specifications.

lisp default: nolisp

Autoindent indents appropriately for *lisp* code, and the ( ) { } [[and]] commands in *open* and *visual* are modified to have meaning for *lisp*.

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. ‡‡

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*.

‡‡ Version 3 only.

remap default: remap

If on, macros are repeatedly tried until they are unchanged. ‡‡ For example, if **o** is mapped to O, and O is mapped to I, then if *remap* is set, **o** will map to I, but if *noremap* is set, it will map to O.

report default: report=5†

Specifies a threshold for feedback from commands. Any command which modifies more than the specified number of lines will provide feedback as to the scope of its changes. For commands such as *global*, *open*, *undo*, and *visual* which have potentially more far reaching scope, the net change in the number of lines in the buffer is presented at the end of the command, subject to this same threshold. Thus notification is suppressed during a *global* command on the individual commands performed.

scroll default: scroll=½ window

Determines the number of logical lines scrolled when an end-of-file is received from a terminal input in command mode, and the number of lines printed by a command mode *z* command (double the value of *scroll*).

sections default: sections=SHNHH HU

Specifies the section macros for the [[and]] operations in *open* and *visual*. The pairs of characters in the options's value are the names of the macros which start paragraphs.

shell, sh default: sh=/bin/sh

Gives the path name of the shell forked for the shell escape command '!', and by the *shell* command. The default is taken from SHELL in the environment, if present.

shiftwidth, sw default: sw=8

Gives the width a software tab stop, used in reverse tabbing with ^D when using *autoindent* to append text, and by the shift commands.

showmatch, sm default: nosm

In *open* and *visual* mode, when a) or } is typed, move the cursor to the matching (or { for one second if this matching character is on the screen. Extremely useful with *lisp*.

slowopen, slow terminal dependent

Affects the display algorithm used in *visual* mode, holding off display updating during input of new text to improve throughput when the terminal in use is both slow and unintelligent. See *An Introduction to Display Editing with Vi* for more details.

tabstop, ts default: ts=8

The editor expands tabs in the input file to be on *tabstop* boundaries for the purposes of display.

taglength, tl default: tl=0

Tags are not significant beyond this many characters. A value of zero (the default) means that all characters are significant.

---

‡‡ Version 3 only.
† 2 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 (even in version 2) files called tags are searched for in the current directory and in /usr/lib (a master file for the entire system.)

term 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.

wrapscan, ws default: ws

Searches using the regular expressions in addressing will wrap around past the end of the file.

wrapmargin, wm default: wm=0

Defines a margin for automatic wrapover of text during input in *open* and *visual* modes. 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.

‡‡ Version 3 only.

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.

This edition documents Version 2 of *edit* and *ex*.

We welcome comments and suggestions about this tutorial and the UNIX documentation in general.

August 31, 1980

†UNIX is a trademark of Bell Laboratories.

Edit: A Tutorial

Ricki Blau

James Joyce

Computing Services
University of California
Berkeley, California 94720

Text editing using a terminal connected to a computer allows one to create, modify, and print text easily. A specialized computer program, known as a *text editor*, assists in creating and revising text. Creating text is very much like typing on an electric typewriter. Modifying text involves telling the text editor what to add, change, or delete. Text is printed by giving a command to print the file contents, with or without special instructions as to the format of the desired output.

These lessons assume no prior familiarity with computers or with text editing. They consist of a series of text editing sessions which will lead you through the fundamental steps of creating and revising a file of text. After scanning each lesson and before beginning the next, you should follow the examples at a terminal to get a feeling for the actual process of text editing. Set aside some time for experimentation, and 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 "Communicating with UNIX" or one of the other tutorials which provide a general introduction to the system. You will be ready to proceed with this lesson as soon as you are familiar with your terminal and its special keys, the login procedure, 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 which the computer performs in order to accomplish a specific task. As an example, a series of steps to balance your checkbook is a program.

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 which you will be learning to use, 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, log out, 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, and another might contain a very long document or program. The only way to save information from one session to the next is to write it to a file, storing it for later use.

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 the coating on magnetic recording tape, on which information is recorded.

buffer — A temporary work space, made available to the user for the duration of a session of text editing and used for building and modifying the text file. We can imagine the buffer as a blackboard that is erased after each class, where each session with the editor is a class.

Session 1: Creating a File of Text

To use the editor you must first make contact with the computer by logging in to UNIX. We'll quickly review the standard UNIX login procedure.

If the terminal you are using is directly linked to the computer, turn it on and press carriage return, usually labelled "RETURN". If your terminal connects with the computer over a telephone line, turn on the terminal, dial the system access number, and, when you hear a high-pitched tone, place the receiver of the telephone in the acoustic coupler. Press carriage return once and await the login message:

login:

Type your login name, which identifies you to UNIX, on the same line as the login message, and press carriage return. If the terminal you are using has both upper and lower case, be sure you enter your login name in lower case; otherwise UNIX assumes your terminal has only upper case and will not recognize lower case letters you may type. UNIX types "login:" and you reply with your login name, for example "susan":

login: **susan** *(and press carriage return)*

(In the examples, input typed by the user appears in **bold face** to distinguish it from the responses from UNIX.)

UNIX will next respond with a request for a password as an additional precaution to prevent unauthorized people from using your account. The password will not appear when you type it to prevent others from seeing it. The message is:

Password: *(type your password and press carriage return)*

If any of the information you gave during the login sequence was mistyped or incorrect, UNIX will respond with

Login incorrect.
login:

in which case you should start the login process anew. Assuming that you have successfully logged in, UNIX will print the message of the day and eventually will present you with a % at the beginning of a fresh line. The % is the UNIX prompt symbol which tells you that UNIX is ready to accept a command.

Asking for *edit*

You are ready to tell UNIX that you want to work with edit, the text editor. Now is a convenient time to choose a name for the file of text which you are about to create. To begin your editing session type **edit** followed by a space and then the filename which you have selected, for example "text". When you have completed the command, press carriage return and wait for edit's response:

```
% edit text    (followed by a carriage 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. It also checked to see if the file you named, "text", already existed. As we expected, it was unable to find such a file since "text" is the name of the new file that we will create. Edit confirms this with the line:

```
"text" No such file or directory
```

On the next line appears edit's prompt ":", announcing that edit expects a command from you. You may now begin to create the new file.

The "not found" message

If you misspelled edit by typing, say, "editor", your request would be handled as follows:

```
% editor
editor: 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, so you may 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 to enter text into the buffer. This is done by *appending* 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, you are creating text. Most edit commands have two forms: a word which describes what the command does and a shorter abbreviation of that word. Either form may be used. Many beginners find the full command names easier to remember, but once you are familiar with editing you may prefer to type the shorter abbreviations. The command to input text is "append" which may be abbreviated "a". Type **append** and press carriage return.

```
% 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:

```
:add
add: Not an editor command
:
```

When you receive a diagnostic message, check what you typed in order to determine what part of your command confused edit. The message above means that edit was unable to recognize your mistyped command and, therefore, did not execute it. Instead, a new ":" appeared to let you know that edit is again ready to receive a command.

Text input mode

By giving the command "append" (or using the abbreviation "a"), you entered *text input mode*, also known as *append mode*. When you enter text input mode, edit responds by doing nothing. You will not receive any prompts while in text input mode. This is your signal that you are to begin entering lines of text. You can enter pretty much anything you want on the lines. The lines are transmitted one by one to the buffer and held there during the editing session. You may append as much text as you want, and *when you wish to stop entering text lines you should type a period as the only character on the line and press carriage return.* When you give this signal that you want to stop appending text, you will exit from text input mode and reenter command mode. Edit will again prompt you for a command by printing ":".

Leaving append mode does not destroy the text in the buffer. You have to leave append mode to do any of the other kinds of editing, such as changing, adding, or printing text. If you type a period as the first character and type any other character on the same line, edit will believe you want to remain in append mode and will not let you out. As this can be very frustrating, be sure to type **only** the period and carriage return.

This is as good a place as any to learn an important lesson about computers and text: a blank space is a character as far as a computer is concerned. If you so much as type a period followed by a blank (that is, type a period and then the space bar on the keyboard), you will remain in append mode with the last line of text being:

```
.
```

Let's say that the lines of text you enter are (try to type **exactly** what you see, including "thiss"):

```
This is some sample text.
And thiss is some more text.
Text editing is strange, but nice.
.
```

The last line is the period followed by a carriage return that gets you out of append mode. If while typing the line you hit an incorrect key, recall that you may delete the incorrect character or cancel the entire line of input by erasing in the usual way. Refer to "Communicating with UNIX" if you need to review the procedures for making a correction. Erasing a character or cancelling a line must be done before the line has been completed by a carriage return. We will discuss changes in lines already typed in session 2.

Writing text to disk

You are now ready to edit the text. The simplest kind of editing is to write it 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. Thus, 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 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 which were written to disk will still be in the buffer, should you want to modify or add to them.

Edit must have a filename to use before it can write a file. If you forgot to indicate the name of the file when you began the editing session, edit will print:

```
No current filename
```

in response to your write command. 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 carriage 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 a "control d". This is done by holding down the control key (usually labelled "CTRL") and simultaneously pressing the d key. This will end your session with UNIX and will ready the terminal for the next user. It is always important to type a "control-d" 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.

Session 2

Login with UNIX as in the first session:

```
login: susan  (carriage return)
Password:     (give password and carriage return)
%
```

This time when you say that you want to edit, you can specify the name of the file you worked on last time. This will start edit working and it will fetch the contents of the file into the buffer, so that you can resume editing the same file. When edit has copied the file into the buffer, it will repeat its name and tell you the number of lines and characters it contains. Thus,

```
% edit text
"text" 3 lines, 90 characters
:
```

means you asked edit to fetch the file named "text" for editing, causing it to copy the 90 characters of text into the buffer. Edit awaits your further instructions. In this session, we will append more text to our file, print the contents of the buffer, and learn to change the text of a line.

Adding more text to the file

If you want to add more to the end of your text you may do so by using the append command to enter text input mode. Here we'll use the abbreviation for the append command, "a".

```
:a
This is text added in Session 2.
It doesn't mean much here, but
it does illustrate the editor.
.
```

Interrupt

Should you press the RUBOUT key (sometimes labelled DELETE) while working with edit, it will send this message to you:

```
Interrupt
:
```

Any command that edit might be executing is terminated by rubout or delete, causing edit to prompt you for a new command. If you are appending text at the time, you will exit from append mode and be expected to give another command. The line of text that you were typing when the append command was interrupted will not be entered into the buffer.

Making corrections

If you have read a general introduction to UNIX, such as "Communicating with UNIX", you will recall that it is possible to erase individual letters that you have typed. This is done by typing the designated erase character, usually the number sign (#), as many times as there are characters you want to erase. If you make a bad start in a line and would like to begin again, this technique is cumbersome — what if you had 15 characters in your line and wanted to get rid of them? To do so either requires:

```
This is yukky tex###############
```

with no room for the great text you'd like to type, or,

```
This is yukky tex@This is great text.
```

When you type the at-sign (@), you erase the entire line typed so far. You may immediately

begin to retype the line. This, unfortunately, does not help after you type the line and press carriage return. To make corrections in lines which have been completed, it is necessary to use the editing commands covered in this session and those that follow.

Listing what's in the buffer

Having appended text to what you wrote in Lesson 1, you might be curious to see what is 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 "$" is a special symbol designating the last line of the buffer, and "p" (or **print**) is the command to print from line 1 to the end of the buffer. Thus, "1,$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 enter into the buffer a character which can't be printed, which is done by striking a key while the CTRL key is depressed. 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 accidently holding down the CTRL key while typing "a". Edit would display

```
it does illustr^Ate the editor.
```

if you asked to have the line printed. 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 which 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, as suggested. 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 carriage return edit is instructed to search for "thiss". If we asked edit to look for a pattern of characters which it could not find in the buffer, it would 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 which it just printed, ready to make a change in the line

The current line

At all times during an editing session, edit keeps track of the line in the buffer where it is positioned. In general, the line which has been most recently printed, entered, or changed is considered to be the current position in the buffer. You can refer to your current position in

the buffer by the symbol period (.) usually known by the name "dot". If you type "." and carriage return you will be instructing edit to print the current line:

```
:.
And thiss is some more text.
```

If you want to know the number of the current line, you can type .= and carriage return, and edit will respond with the line number:

```
:.=
2
```

If you type the number of any line and a carriage return, edit will position you at that line and print its contents:

```
:2
And thiss is some more text.
```

You should experiment with these commands to assure yourself that you understand what they do.

Numbering lines (nu)

The **number (nu)** command is similar to print, giving both the number and the text of each printed line. To see the number and the text of the current line type

```
:nu
     2  And thiss is some more text.
```

Notice that the shortest abbreviation for the number command is "nu" (and not "n" which is used for a different command). You may specify a range of lines to be listed by the number command in the same way that lines are specified for print. For example, "1,$nu" lists all lines in the buffer with the corresponding line numbers.

Substitute command (s)

Now that we have found our misspelled word it is time to change it from "thiss" to "this". As far as edit is concerned, changing things is a matter of substituting one thing for another. As *a* stood for *append*, so *s* stands for *substitute*. We will use the abbreviation "s" to reduce the chance of mistyping the substitute command. This command will instruct edit to make the change:

```
2s/thiss/this/
```

We first indicate the line to be changed, line 2, and then type an "s" to indicate we want substitution. Inside the first set of slashes are the characters that we want to change, followed by the characters to replace them and then a closing slash mark. To summarize:

2s/ what is to be changed / what to change to /

If edit finds an exact match of the characters to be changed it will make the change only in the first occurrence of the characters. If it does not find the characters to be changed it will respond:

```
Substitute pattern match failed
```

indicating your instructions could not be carried out. When edit does find the characters which you want to change, it will make the substitution and automatically print the changed line, so that you can check that the correct subsutution 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 positioned ("."). In this case, the command without a line number would have produced the same result because we were already positioned at the line we wished to change.

For another illustration of substitution we may choose the line:

Text editing is strange, but nice.

We might like to be a bit more positive. Thus, we could take out the characters "strange, but " so the line would read:

Text editing is nice.

A command which will first position edit at that 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. This illustrates that we do not necessarily have to use line numbers to identify a line to edit. Instead, we may identify the line we want to change by asking edit to search for a specified pattern of letters which occurs in that line. The parts of the above command are:

/strange/	tells edit to find the characters "strange" in the text
s	tells edit we want to make a substitution
/strange, but //	substitutes nothing at all for the characters "strange, but "

You should note the space after "but" in "/strange, but /". If you do not indicate the space is to be taken out, your line will be:

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 can 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, give the command

```
:z
```

If no starting line number is given for the z command, printing will start at the "current" line, in this case the last line printed. Viewing lines in the buffer one screen full at a time is known as paging. Paging can also be used to print a section of text on a hard-copy terminal.

Saving the modified text

This seems to be a good place to pause in our work, and so we should end the second session. If you (in haste) type "q" to quit the session your dialogue with edit will be:

```
:q
No write since last change (q! quits)
:
```

This is edit's warning that you have not written the modified contents of the buffer to disk. You run the risk of losing the work you have done during the editing session since the latest write command. Since in this lesson we have not written to disk at all, everything we have done would be lost. If we did not want to save the work done during this editing session, we would have to type "q!" to confirm that we indeed wanted to end the session immediately, losing the contents of the buffer. However, since we want to preserve what we have edited, we need to say:

```
:w
"text" 6 lines, 171 characters
```

and then,

```
:q
% (control d)
```

and hang up the phone or turn off the terminal when UNIX asks for a login name. This is the end of the second session on UNIX text editing.

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 say

```
% 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 saying:

```
: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:

```
:2,4m$
```

This command 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. Thus,

```
:1,6m20
```

would instruct edit to move lines 1 through 6 (inclusive) to a position after line 20 in the buffer. To move only one line, say, line 4, to a position in the buffer after line 6, the command would be "4m6".

Let's move some text using the command:

```
:5,$m1
2 lines moved
it does illustrate the editor.
```

After executing a command which changes more than one line of the buffer, edit tells how many lines were affected by the change. The last moved line is printed for your inspection. If you want to see more than just the last line, 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.
```

We can restore the original order by typing:

```
:4,$m1
```

or, combining context searching and the move command:

```
:/And this is some/,/This is text/m/This is some sample/
```

The problem with combining context searching with the move command is that the chance of making a typing error in such a long command is greater than if one types line numbers.

Copying lines (copy)

The **copy** command is used to make a second copy of specified lines, leaving the original lines where they were. Copy has the same format as the move command, for example:

```
:12,15copy $
```

makes a copy of lines 12 through 15, placing the added lines after the buffer's end ($). Experiment with the copy command so that you can become familiar with how it works. Note that the shortest abbreviation for copy is "co" (and **not** the letter "c" which has another meaning).

Deleting lines (d)

Suppose you want to delete the line

```
This is text added in Session 2.
```

from the buffer. If you know the number of the line to be deleted, you can type that number followed by "**delete**" or "**d**". This example deletes line 4:

```
:4d
It doesn't mean much here, but
```

Here "4" is the number of the line to be deleted and "delete" or "d" is the command to delete the line. After executing the delete command, edit prints the line which has become the current line (".").

If you do not happen to know the line number you can search for the line and then delete it using this sequence of commands:

```
:/added in Session 2./
This is text added in Session 2.
:d
It doesn't mean much here, but
```

The "/added in Session 2./" asks edit to locate and print the next line which contains the indicated text. Once you are sure that you have correctly specified the line that you want to delete, you can enter the delete (d) command. In this case it is not necessary to specify a line number before the "d". If no line number is given, edit deletes the current line ("."), that is, the line found by our search. After the deletion, your buffer should contain:

```
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.
```

To delete both lines 2 and 3:

```
And this is some more text.
Text editing is nice.
```

you type

```
:2,3d
```

which specifies the range of lines from 2 to 3, and the operation on those lines — "d" for delete.

Again, this presumes 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 as so:

```
:/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 carriage 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. 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 such as print which do not change the buffer. Most importantly, the **only** command which can be reversed by undo is the last "undo-able" command which you gave.

To illustrate, let's issue an undo command. Recall that the last buffer-changing command we gave deleted the lines which were formerly numbered 2 and 3. Executing undo at this moment will reverse the effects of the deletion, causing those two lines to be replaced in the buffer.

```
:u
2 more lines in file after undo
And this is some more text.
```

Here again, edit informs you if the command affects more than one line, and prints the text of the line which is now "dot" (the current line).

More about the dot (.) and buffer end ($)

The function assumed by the symbol dot depends on its context. It can be used:

1. to exit from append mode we type dot (and only a dot) on a line and press carriage return;
2. to refer to the line we are at in the buffer.

Dot can also be combined with the equal sign to get the number of the line currently being edited:

```
:.=
```

Thus if we type ".=" we are asking for the number of the line and if we type "." we are asking for the text of the line.

In this editing session and the last, we used the dollar sign to indicate the end of the buffer in commands such as print, copy, and move. The dollar sign as a command asks edit to print the last line in the buffer. If the dollar sign is combined with the equal sign ($=) edit will print the line number corresponding to the last line in the buffer.

"." and "$" therefore represent line numbers. Whenever appropriate, these symbols can be used in place of line numbers in commands. For example

```
:.,$d
```

instructs edit to delete all lines from the current line (.) to the end of the buffer.

Moving around in the buffer (+ and −)

It is frequently convenient during an editing session to go back and re-read a previous line. We could specify a context search for a line we want to read if we remember some of its text, but if we simply want to see what was written a few, say 3, lines ago, we can type

```
−3p
```

This tells edit to move back to a position 3 lines before the current line (.) and print that line. We can move forward in the buffer similarly:

```
+2p
```

instructs edit to print the line which is 2 ahead of our current position.

You may use "+" and "−" in any command where edit accepts line numbers. Line numbers specified with "+" or "−" can be combined to print a range of lines. The command

```
:−1,+2copy$
```

makes a copy of 4 lines: the current line, the line before it, and the two after it. The copied lines will be placed after the last line in the buffer ($).

Try typing only "−"; you will move back one line just as if you had typed "−1p". Typing the command "+" works similarly. You might also try typing a few plus or minus signs in a row (such as "+++") to see edit's response. Typing a carriage return alone on a line is the equivalent of typing "+1p"; it will move you one line ahead in the buffer and print that line.

If you are at the last line of the buffer and try to move further ahead, perhaps by typing a "+" or a carriage return alone on the line, edit will remind you that you are at the end of the buffer:

```
At end-of-file
```

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
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)

There may be occasions when you want to 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 in order to accept the text which will replace them. Let's say we 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 can 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 a carriage return enters 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.

Session 4

This lesson covers several topics, starting with commands which apply throughout the buffer, characters with special meanings, and how to issue UNIX commands while in the editor. The next topics deal with files: more on reading and writing, and methods of recovering files lost in a crash. The final section suggests sources of further information.

Making commands global (g)

One disadvantage to the commands we have used for searching or substituting is that if you have a number of instances of a word to change it appears that you have to type the command repeatedly, once for each time the change needs to be made. Edit, however, provides a way to make commands apply to the entire contents of the buffer — the **global** (**g**) command.

To print all lines containing a certain sequence of characters (say, "text") the command is:

```
:g/text/p
```

The "g" instructs edit to make a global search for all lines in the buffer containing the characters "text". The "p" prints the lines found.

To issue a global command, start by typing a "g" and then a search pattern identifying the lines to be affected. Then, on the same line, type the command to be executed on the identified lines. Global substitutions are frequently useful. For example, to change all instances of the word "text" to the word "material" the command would be a combination of the global search and the substitute command:

```
:g/text/s/text/material/g
```

Note the "g" at the end of the global command which instructs edit to change each and every instance of "text" to "material". If you do not type the "g" at the end of the command only the *first* instance of "text" in each line will be changed (the normal result of the substitute command). The "g" at the end of the command is independent of the "g" at the beginning. You may give a command such as:

```
:14s/text/material/g
```

to change every instance of "text" in line 14 alone. Further, neither command will change "Text" to "material" because "Text" begins with a capital rather than a lower-case *t*.

Edit does not automatically print the lines modified by a global command. If you want the lines to be printed, type a "p" at the end of the global command:

```
:g/text/s/text/material/gp
```

The usual qualification should be made about using the global command in combination with any other — in essence, be sure of what you are telling edit to do to the entire buffer. For example,

```
:g/ /d
72 less lines in file after global
```

will delete every line containing a blank anywhere in it. This could adversely affect your document, since most lines have spaces between words and thus would be deleted. After executing the global command, edit will print a warning if the command added or deleted more than one line. Fortunately, the undo command can reverse the effects of a global command. You should experiment with the global command on a small buffer of text to see what it can do for you.

More about searching and substituting

In using slashes to identify a character string that we want to search for or change, we have always specified the exact characters. There is a less tedious way to repeat the same string of characters. To change "noun" to "nouns" we may type either

```
:/noun/s/noun/nouns/
```

as we have done in the past, or a somewhat abbreviated command:

```
:/noun/s//nouns/
```

In this example, the characters to be changed are not specified — there are no characters, not even a space, between the two slash marks which 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.
```

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 character string.

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 which occur at the end of a line.

```
:g/ing$/s//ed/p
```

tells the editor to search for all lines ending in "ing" (and nothing else, not even a blank space), to change each final "ing" to "ed" 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 temporarily lose its special significance by typing another special character, the backslash (\), before it.

```
:s/\$/dollar/
```

looks for the character "$" in the current line and replaces it by the word "dollar". Were it not for the backslash, the "$" would have represented "the end of the line" in your search, rather than the character "$". The backslash retains its special significance unless it is preceded by another backslash.

Issuing UNIX commands from the editor

After creating several files with the editor, you may want to delete files no longer useful to you or ask for a list of your files. Removing and listing files are not functions of the editor, and so they require the use of UNIX system commands (also referred to as "shell" commands, as "shell" is the name of the program that processes UNIX commands). You do not need to quit the editor to execute a UNIX command as long as you indicate that it is to be sent to the shell for execution. To use the UNIX command *rm* to remove the file named "junk" type:

```
:!rm junk
!
:
```

The exclamation mark (!) indicates that the rest of the line is to be processed as a UNIX command. If the buffer contents have not been written since the last change, a warning will be printed before the command is executed. The editor prints a "!" when the command is completed. The tutorial "Communicating with UNIX" describes useful features of the system, of which the editor is only one part.

Filenames and file manipulation

Throughout each editing session, edit keeps track of the name of the file being edited as the *current filename*. Edit remembers as the current filename the name given when you entered the editor. The current filename changes whenever the edit (e) command is used to specify a new file. Once edit has recorded a current filename, it inserts that name into any command where a filename has been omitted. If a write command does not specify a file, edit, as we have seen, supplies the current filename. You can have the editor write onto a different file by including its name in the write command:

```
:w chapter3
"chapter3" 283 lines, 8698 characters
```

The current filename remembered by the editor *will not be changed as a result of the write command unless it is the first filename given in the editing session.* Thus, in the next write command which does not specify a name, edit will write onto the current file and not onto the file "chapter3".

The file (f) command

To ask for the current filename, type file (or f). In response, the editor provides current information about the buffer, including the filename, your current position, and the number of lines in the buffer:

```
:f
"text" [Modified] line 3 of 4 --75%--
```

If the contents of the buffer have changed since the last time the file was written, the editor will tell you that the file has been "[Modified]". After you save the changes by writing onto a disk file, the buffer will no longer be considered modified:

```
: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 without destroying the text already there. To use it, specify the line after which the new text will be

placed, the command *r*, and then the name of the file.

```
:Sr bibliography
"bibliography" 18 lines, 473 characters
```

This command reads in the file *bibliography* and adds it to the buffer after the last line. The current filename is not changed by the read command unless it is the first filename given in the editing session.

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,Sw 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.

Recovering files

Under most circumstances, edit's crash recovery mechanism is able to save work to within a few lines of changes after a crash or if the phone is hung up accidently. If you lose the contents of an editing buffer in a system crash, you will normally receive mail when you login which 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.

```
:recover chapó
```

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.

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
```

and then seek help. Do not simply leave the editor. If you do, the buffer will be lost, and you may not be able to save your file. After a preserve, you can use the recover command once the problem has been corrected.

If you make an undesirable change to the buffer and issue a write command before discovering your mistake, the modified version will replace any previous version of the file. Should you ever lose a good version of a document in this way, do not panic and leave the editor. As long as you stay in the editor, the contents of the buffer remain accessible. Depending on the nature of the problem, it may be possible to restore the buffer to a more complete state with the undo command. After fixing the damaged buffer, you can again write the file to disk.

Further reading and other information

Edit is an editor designed for beginning and casual users. It is actually a version of a more powerful editor called *ex*. These lessons are intended to introduce you to the editor and its more commonly-used commands. We have not covered all of the editor's commands, just a selection of commands which should be sufficient to accomplish most of your editing tasks. You can find out more about the editor in the *Ex Reference Manual*, which is applicable to both *ex* and *edit*. The manual is available from the Computer Center Library, 218 Evans Hall. One way to become familiar with the manual is to begin by reading the description of commands that you already know.

Using *ex*

As you become more experienced with using the editor, you may still find that edit continues to meet your needs. However, should you become interested in using ex, it is easy to switch. To begin an editing session with ex, use the name ex in your command instead of edit.

Edit commands work the same way in ex, but the editing environment is somewhat different. You should be aware of a few differences that exist between the two versions of the editor. In edit, only the characters "^", "$", and "\" have special meanings in searching the buffer or indicating characters to be changed by a substitute command. Several additional characters have special meanings in ex, as described in the *Ex Reference Manual*. Another feature of the edit environment prevents users from accidently entering two alternative modes of editing, *open* and *visual*, in which the editor behaves quite differently than in normal command mode. If you are using ex and the editor behaves strangely, you may have accidently entered open mode by typing "o". Type the ESC key and then a "q" to get out of open or visual mode and back into the regular editor command mode. The document *An Introduction to Display Editing with Vi* provides a full discussion of visual mode.

This tutorial was produced at the Computer Center of the University of California, Berkeley. We welcome comments and suggestions concerning this item and the UNIX documentation in general.

Index

Ex/Edit Command Summary (Version 2.0)

Ex and *edit* are text editors, used for creating and modifying files of text on the UNIX computer system. *Edit* is a variant of *ex* with features designed to make it less complicated to learn and use. In terms of command syntax and effect the editors are essentially identical, and this command summary applies to both.

The summary is meant as a quick reference for users already acquainted with *edit* or *ex*. Fuller explanations of the editors may be found in the documents *Edit: A Tutorial* (a self-teaching introduction) and the *Ex Reference Manual* (the comprehensive reference source for both *edit* and *ex*). Both of these writeups are available in the Computing Services Library.

In the examples included with the summary, commands and text entered by the user are printed in **boldface** to distinguish them from responses printed by the computer.

The Editor Buffer

In order to perform its tasks the editor sets aside a temporary work space, called a *buffer*, separate from the user's permanent file. Before starting to work on an existing file the editor makes a copy of it in the buffer, leaving the original untouched. All editing changes are made to the buffer copy, which must then be written back to the permanent file in order to update the old version. The buffer disappears at the end of the editing session.

Editing: Command and Text Input Modes

During an editing session there are two usual modes of operation: *command* mode and *text input* mode. (This disregards, for the moment, *open* and *visual* modes, discussed below.) In command mode, the editor issues a colon prompt (:) to show that it is ready to accept and execute a command. In text input mode, on the other hand, there is no prompt and the editor merely accepts text to be added to the buffer. Text input mode is initiated by the commands *append*, *insert*, and *change*. It is terminated by typing a period as the first character on a line, followed immediately by a carriage return.

Line Numbers and Command Syntax

The editor keeps track of lines of text in the buffer by numbering them consecutively starting with 1 and renumbering as lines are added or deleted. At any given time the editor is positioned at one of these lines; this position is called the *current line*. Generally, commands that change the contents of the buffer print the new current line at the end of their execution.

Most commands can be preceded by one or two line-number addresses which indicate the lines to be affected. If one number is given the command operates on that line only; if two, on an inclusive range of lines. Commands that can take line-number prefixes also assume default prefixes if none are given. The default assumed by each command is designed to make it convenient to use in many instances without any line-number prefix. For the most part, a command used without a prefix operates on the current line, though exceptions to this rule should be noted. The *print* command by itself, for instance, causes one line, the current line, to be printed at the terminal.

The summary shows the number of line addresses that can be prefixed to each command as well as the defaults assumed if they are omitted. For example, *(.,.)* means that up to 2 line-numbers may be given, and that if none is given the command operates on the current line. (In the address prefix notation, "." stands for the current line and "$" stands for the last line of the buffer.) If no such notation appears, no line-number prefix may be used.

Some commands take trailing information; only the more important instances of this are mentioned in the summary.

Open and Visual Modes

Besides command and text input modes, *ex* and *edit* provide on some terminals other modes of editing, *open* and *visual*. In these modes the cursor can be moved to individual words or characters in a line. The commands are then given are very different from the standard editor commands; most do not appear on the screen when typed. *An Introduction to Display Editing with Vi* provides a full discussion.

Special Characters

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

. & * [ ] ~

To use one of the special characters as its simple graphic representation rather than with its special meaning, precede it by a backslash (\). The backslash always has a special meaning. Consult the more complete writeups on *edit* and *ex* for details on the use of special characters.

Name	Abbr	Description	Examples
(.)**append**	**a**	Begins text input mode, adding lines to the buffer after the line specified. Appending continues until "." is typed alone at the beginning of a new line, followed by a carriage return. 0a places lines at the beginning of the buffer.	**:a** **Three lines of text** **are added to the buffer** **after the current line.** **.** :
(.,.)**change**	**c**	Deletes indicated line(s) and initiates text input mode to replace them with new text which follows. New text is terminated the same way as with *append*.	**:5,6c** **Lines 5 and 6 are** **deleted and replaced by** **these three lines.** **.** :
(.,.)**copy** *addr*	**co**	Places a copy of the specified lines after the line indicated by *addr*. The example places a copy of lines 8 through 12, inclusive, after line 25.	**:8,12co 25** Last line copied is printed :
(.,.)**delete**	**d**	Removes lines from the buffer and prints the current line after the deletion.	**:13,15d** New current line is printed :
edit *file* **edit!** *file*	**e** **e!**	Clears the editor buffer and then copies into it the named *file*, which becomes the current file. This is a way of shifting to a different file without leaving the editor. The editor issues a warning message if this command is used before saving changes made to the file already in the buffer; using the form e! overrides this protective mechanism.	**:e ch10** No write since last change **:e! ch10** "ch10" 3 lines, 62 characters :
file *name*	**f**	If followed by a *name*, renames the current file to *name*. If used without *name*, prints the name of the current file.	**:f ch9** "ch9" [Modified] 3 lines ... **:f** "ch9" [Modified] 3 lines ... :
(1,$)**global** (1,$)**global!**	**g** **g!** or **v**	**global** */pattern/ commands* Searches the entire buffer (unless a smaller range is specified by line-number prefixes) and executes *commands* on every line with an expression matching *pattern*. The second form, abbreviated either g! or v, executes *commands* on lines that *do not* contain the expression *pattern*.	**:g/nonsense/d** :
(.)**insert**	**i**	Inserts new lines of text immediately before the specified line. Differs from *append* only in that text is placed before, rather than after, the indicated line. In other words, 1i has the same effect as 0a.	**:1i** **These lines of text will** **be added prior to line 1.** **.** :
(.,.+1)**join**	**j**	Join lines together, adjusting white space (spaces and tabs) as necessary.	**:2,5j** Resulting line is printed :
(.,.)**list**	**l**	Prints lines in a more unambiguous way than the *print* command does. The end of a line, for example, is marked with a "$", and tabs printed as "^I".	**:9l** This is line nine$:

Name	Abbr	Description	Examples
(...)**move** *addr*	**m**	Moves the specified lines to a position after the line indicated by *addr*.	:12,15m 25 New current line is printed :
(...)**number**	**nu**	Prints each line preceded by its buffer line number.	:**nu** 10 This is line ten :
(.)**open**	**o**	Too involved to discuss here, but if you enter open mode accidentally, press the ESC key followed by q to get back into normal editor command mode. *Edit* is designed to prevent accidental use of the open command.	
preserve	**pre**	Saves a copy of the current buffer contents as though the system had just crashed. This is for use in an emergency when a *write* command has failed and you don't know how else to save your work. Seek assistance from a consultant as soon as possible after saving a file with the *preserve* command, because the file is saved for only one week.	:**preserve** File preserved. :
(...)**print**	**p**	Prints the text of line(s).	:**+2,+3p** The second and third lnes after the current line :
quit **quit!**	**q** **q!**	Ends the editing session. You will receive a warning if you have changed the buffer since last writing its contents to the file. In this event you must either type **w** to write, or type **q!** to exit from the editor without saving your changes.	:**q** No write since last change :**q!** %
(.)**read** *file*	**r**	Places a copy of *file* in the buffer after the specified line. Address 0 is permissible and causes the copy of *file* to be placed at the beginning of the buffer. The *read* command does not erase any text already in the buffer. If no line number is specified, *file* is placed after the current line.	:**0r newfile** "newfile" 5 lines, 86 characters :
recover *file*	**rec**	Retrieves a copy of the editor buffer after a system crash, editor crash, phone line disconnection, or *preserve* command.	
set *parameter*	**se**	Changes the settings of one or more editor options; lists the current settings of options which have been changed from their defaults; or lists the settings of all options. For more details consult the manual.	
(...)**substitute**	**s**	**substitute**/*pattern*/*replacement*/ **substitute**/*pattern*/*replacement*/**gc** Replaces the first ocurrence of *pattern* on a line with *replacement*. Including a **g** after the command changes all occurrences of *pattern* on the line. The **c** option allows the user to confirm each substitution before it is made; see the manual for details.	:**3p** Line 3 contains a misstake :**s/misstake/mistake/** Line 3 contains a mistake :

Name	Abbr	Description	Examples
undo	u	Reverses the changes made in the buffer by the last buffer-editing command. Note that this example contains a notification about the number of lines affected.	:1,15d 15 lines deleted new line number 1 is printed :u 15 more lines in file ... old line number 1 is printed :
(1,S)write *file* (1,S)write! *file*	w w!	Copies data from the buffer onto a permanent file. If no *file* is named, the current filename is used. The file is automatically created if it does not yet exist. A response containing the number of lines and characters in the file indicates that the write has been completed successfully. The editor's built-in protections against overwriting existing files will in some circumstances inhibit a write. The form w! forces the write, confirming that an existing file is to be overwritten.	:w "file7" 64 lines, 1122 characters :w file8 "file8" File exists ... :w! file8 "file8" 64 lines, 1122 characters :
(.)z *count*	z	Prints a screen full of text starting with the line indicated; or, if *count* is specified, prints that number of lines. Variants of the *z* command are described in the manual.	
!*command*		Executes the remainder of the line after ! as a UNIX command. The buffer is unchanged by this, and control is returned to the editor when the execution of *command* is complete.	:!date Fri Jun 9 12:15:11 PDT 1978 ! :
control-d		Prints the next *scroll* of text, normally half of a screen. See the manual for details of the *scroll* option.	
(.+1)<cr>		An address alone followed by a carriage return causes the line to be printed. A carriage return by itself prints the line following the current line.	:<cr> the line after the current line :
/*pattern*/		Searches for the next line in which *pattern* occurs and prints it.	:/This pattern/ This pattern next occurs here. :
//		Repeats the most recent search.	:// This pattern also occurs here. :
?*pattern*?		Searches in the reverse direction for *pattern*.	
??		Repeats the most recent search, moving in the reverse direction through the buffer.	

MAIL REFERENCE MANUAL

Kurt Shoens

Version 1.3

December 2, 1979

1. Introduction

Mail provides a friendly environment for sending and receiving mail by dividing incoming mail into its constituent messages and allowing the user to deal with them in any order. It provides a set of *ed*-like commands for manipulating messages and sending mail.

This document describes the Mail command at an introductory level appropriate for the casual user as well as a more detailed level appropriate for those whose usage of Mail is sufficiently frequent to justify such familiarity. The reader is assumed to be familiar with the UNIX[1] Shell, the text editor, and some of the common UNIX commands. If you are a neophyte Mail user, section two of this document should provide enough information to allow you to effectively use Mail. The balance of this document describes more advanced features, which are useful to those commonly barraged with a large volume of mail.

2. Common usage

The Mail command has two distinct usages, according to whether one wants to send or receive mail. Sending mail is simple: to send a message to a user whose login name is, say, "root," use the Shell command:

```
% Mail root
```

then type your message. When you reach the end of the message, type an EOT (control−d) at the beginning of a line, which will cause Mail to echo "EOT" and return you to the Shell. When the user you sent mail to next logs in, he will receive the message:

```
You have mail.
```

to alert him to the existence of your message. Incidentally, once you have sent mail to someone, there is no way to undo the act, so be careful. The message your recipient reads will consist of the message you typed, preceded by a line telling who sent the message (your login name), the teletype from which the message was sent, and the date and time it was sent.

If you want to send the same message to several other people, you can list all of their login names on the command line. Thus,

```
% Mail sam bob john
Tuition fees are due next Friday. Don't forget!!
```

---

[1] UNIX is a trademark of Bell Laboratories.

```
<Control-d>
EOT
%
```

will send the reminder to sam, bob, and john.

If, when you log in, you see the message,

```
You have mail.
```

you can read the mail by typing simply:

```
% Mail
```

Mail will respond by typing its version number and date and then listing the messages you have waiting. Then it will type an underscore and await your command. The messages are assigned numbers starting with 1 — you can refer to the messages with these numbers.

To look at a specific message, use the type command, which may be abbreviated to simply t. For example, if you had the following messages:

```
1 root    Wed Sep 21 09:21  "Tuition fees"
2 sam     Tue Sep 20 22:55
```

you could examine the first message by giving the command:

```
type 1
```

which might cause Mail to respond with, for example:

```
Message 1:
From root  tty8  Wed Sep 21 09:21:45 1978
Subj: Tuition fees

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

Normally, each message you receive is saved in the file *mbox* in your login directory at the time you leave Mail. Often, however, you will not want to save a particular message you have received because it is only of passing interest. To avoid saving a message in *mbox* you can delete it using the delete command. In our example,

```
delete 1
```

will prevent Mail from saving message 1 (from root) in *mbox*. In addition to not saving deleted messages, Mail will not let you type them, either. The effect is to make the message disappear altogether, along with its number. The delete command can be abbreviated to simply d.

When you have perused all of the messages of interest, you can leave Mail with the quit command, which saves all of the messages you have typed but not deleted in the file *mbox* in your login directory. Deleted messages are discarded irretrievably, and messages left untouched are preserved in your system mailbox so that you will see them the next time you type:

```
% Mail
```

The quit command can be abbreviated to simply q.

If you wish for some reason to leave Mail quickly without altering either your system mailbox or *mbox*, you can type the x command (short for exit), which will immediately return you to the Shell without changing anything.

If, instead, you want to execute a Shell command without leaving Mail, you can type the command preceded by an exclamation point, just as in the text editor. Thus, for instance:

```
!date
```

will print the current date without leaving Mail.

Finally, the help command is available to print out a brief summary of the Mail commands, using only the single character command abbreviations.

3. Tilde escapes

While typing in a message to be sent to others, it is often useful to be able to invoke the text editor on the partial message, print the message, execute a shell command, or perform some other auxiliary function. Mail provides these capabilities through *tilde escapes*, which consist of a tilde (~) at the beginning of a line, followed by a single character which indicates the function to be performed. For example, to print the text of the message so far, use:

~p

which will print a line of dashes, the recipients of your message, and the text of the message so far. If you are dissatisfied with the message as it stands, you can invoke the text editor on it using the escape

~e

which causes the message to be copied into a temporary file and an instance of the editor to be spawned. After modifying the message to your satisfaction, write it out and quit the editor. Mail will respond by typing

(continue)

after which you may continue typing text which will be appended to your message, or type <control-d> to end the message.

It is often useful to be able to include the contents of some file in your message; the escape

~r filename

is provided for this purpose, and causes the named file to be appended to your current message. Mail complains if the file doesn't exist or can't be read. If the read is successful, the number of lines and characters appended to your message is printed, after which you may continue appending text.

As a special case of ~r, the escape

~d

reads in the file "dead.letter" in your home directory. This is often useful since Mail copies the text of your message there when you abort a message with RUBOUT.

In order to save the current text of your message on a file you may use the

~w filename

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.

If you are sending mail from within Mail's command mode (read about the **reply** and **mail** commands, section six), 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. 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; in fact, 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 "Subj:" 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 "Subj:" 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 printhead) 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 # 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 "Subj:" 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

~|command

which pipes your message through the command and uses the output as the new text of your message. If the command produces no output, Mail assumes that something is amiss and retains the old version of your message. A frequently-used filter is the command *fmt* which is designed to format outgoing mail.

If you wish (for some reason) to send a message which contains a line beginning with a tilde, you must double it. Thus, for example,

~~This line begins with a tilde.

sends the line

~This line begins with a tilde.

Finally, the escape

~?

prints out a brief summary of the available tilde escapes.

4. Message lists

The type and delete commands described in section two take a list of messages as argument, as do many of the commands described in section six. This section describes the construction of message lists in general.

A *message list* consists of a list of message numbers, ranges, and names, separated by spaces or tabs. Message numbers may be either decimal numbers, which directly specify messages, or one of the special characters "↑" "." or "$" to specify the first relevant, current, or last relevant message, respectively. *Relevant* here means, for most commands "not deleted" and "deleted" for the **undelete** command.

A range of messages consists of two message numbers (of the form described in the previous paragraph) separated by a dash. Thus, to print the first four messages, use

type 1−4

and to print all the messages from the current message to the last message, use

type .−$

A *name* is a user name. All of the user names given in the message list are collected together and each message selected by other means is checked to make sure it was sent by one of the named users. If the message consists entirely of user names, then every message sent by one those users which is *relevant* (in the sense described earlier) is selected. Thus, to print every message sent to you by "root," do

type root

As a shorthand notation, you can specify simply "*" to get every *relevant* (same sense) message. Thus,

type *

prints all undeleted messages,

delete *

deletes all undeleted messages, and

undelete *

undeletes all deleted messages.

5. Command line options

This section describes the alternate usages of Mail from the shell.

As you continue to receive system mail, you will most likely accumulate a large collection of messages in the file *mbox*. In order to help you deal with this, Mail allows you to edit files of messages by using the -f flag. Specifically,

Mail −f filename

causes Mail to edit "filename" and

Mail −f

causes Mail to read "mbox" in your home directory. All of the Mail commands except **preserve** are available to edit the messages. When you type the **quit** command, Mail will write the updated file back.

Since you will usually have a large number of messages stored in *mbox*, Mail will only print out the first 18 message headers when editing more than 18 messages. To display the other message headers, the **headers** command takes as an optional argument either + or − to move forward or back to the next or previous 18 message group.

If you send mail over a noisy phone line, you will notice that many of the garbage characters turn out to be the RUBOUT character, which causes Mail to abort the message. To deal with this annoyance, you can invoke Mail with the -i option to causes these garbage characters to be ignored. Unfortunately, as you are typing in a line of text to a program, the little gnome which gathers up the characters is instructed to throw them all away when a RUBOUT is seen. For this reason, Mail indicates that a RUBOUT has been received by echoing an @ to tell you that everything you had typed on that line has been thrown away.

6. Additional commands

This section describes additional Mail commands available when receiving mail.

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,

type 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.

The **−** command goes to the previous message and prints it. The **−** command may be given a decimal number *n* as an argument, in which case the *n*th previous message is gone to and printed.

The **save** command allows you to save messages received from others on a file other than *mbox*. Its syntax varies somewhat from the other commands which accept a message list in that the final word on the command line is taken to be the file on which to save the messages. The named messages are appended to the file (which is created if it did not already exist) and are marked as saved. Saved messages are not automatically saved in *mbox* at quit time, nor are they selected by the **next** command described above, unless explicitly specified. The **save** command provides a facility for saving messages pertaining to a particular subject or from a particular person in a special place.

The **undelete** command causes a message which had been deleted previously to regain its initial status. Only messages which are already deleted may be undeleted. This command may be abbreviated to **u**.

The **preserve** command takes a message list and marks each message therein so that it will be saved in your system mailbox instead of being deleted or saved in *mbox* when you quit. This is useful for saving messages of importance that you want to see again, or messages not intended for you if you are sharing a login name.

Often, one wants to deal with a message by responding to its author right then and there. The **reply** command is useful for this purpose: it takes a message list and sends mail to the authors of those messages. The message is collected in the usual fashion by reading up to an EOT. All of the tilde escapes described in section three will work in **reply**. Additionally, if there are header fields in the message being replied to, this information is copied into the new message. The **reply** command can be abbreviated to **r**.

In order to simply mail to a user inside of Mail, the **mail** command is provided. This sends mail in the manner described for the **reply** command above, except that the user supplies a list of recipient login names and distribution groups. All of the tilde escapes described in section three will work in **mail**. The **mail** command may be abbreviated to **m**.

In order to edit individual messages using the text editor, the **edit** command is provided. The **edit** command takes a list of message as described under the **type** command and processes each by writing it into the file Message*x* 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**.

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.

When Mail is invoked to receive mail, it prints out the message header for each message. In order to reprint the headers for remaining messages (those which haven't been deleted), you may type the **headers** command. Deleted messages do not appear in the listing, saved messages are flagged with a "*" and preserved messages are flagged with a "P."

The **from** command takes a list of messages and prints out the header lines for each one; hence

from joe

is the easy way to display all the message headers from "joe."

The **top** command takes a message list and prints the first five lines of each addressed message. It may be abbreviated to **to**.

The **dt** command deletes the current message and prints the next message. It is useful for quickly reading and disposing of mail.

7. Summary of commands, escapes, and options

This sections describes tersely all of the Mail commands, escapes, and options. For each command, its most abbreviated form (in brackets) and a short description of the command is given below.

First, message lists are computed by determining the set M which consists of all message referenced explicitly or through ranges. Then, the set U is computed, which consists of all messages sent by *any* of the user names specified. Finally, the message list is calculated by finding the intersection of sets M and U.

Each Mail command is typed on a line by itself, and may take arguments following the command word. The command need not be typed in its entirety — the first command which matches the typed prefix is used. If the argument begins with a digit or special character, then no space is required following the command letter, but otherwise the space is required. If a Mail command does not take arguments, they may be specified, even though they are ignored. For the commands which take message lists as arguments, if no message list is given, then the next message forward which satisfies the command's requirements is used. If there are no messages forward of the current message, the search proceeds backwards, and if there are no good messages at all, Mail types "No applicable messages" and aborts the command.

−	[−] Goes to the previous message and prints it out. If given a numeric argument *n*, goes to the *n*th previous message and prints it. If there is no previous message, it prints "Nonzero address required."
=	[=] Prints out the current message number. Takes no arguments.
?	[?] Prints out the file /usr/lib/Mail.help, which contains a brief summary of the commands. Takes no arguments.
!	[!] Executes the UNIX Shell command which follows. Unlike other commands, there does not need to be a space after the exclamation point.
alias	[a] With no arguments, prints out all currently-defined aliases. With one argument, prints out that alias. With more than one argument, adds the users named in the second and later arguments to the alias named in the first argument.
chdir	[c] Changes the user's working directory to that specified, if given. If no directory is given, then changes to the user's login directory.
delete	[d] Takes a list of messages as argument and marks them all as deleted. Deleted messages will not be saved in *mbox*, nor will they be available for most other commands. Default messages may not be deleted already.
dp	[dp] Deletes the current message and prints the next message. If there is no next message, types out "At EOF."

dt [dt] Same as **dp**.

edit [e] Takes a list of messages and points the text editor at each one in turn. On return from the editor, the message is read back in. The default message for edit may not be saved or deleted.

exit [ex] Effects an immediate return to the Shell without modifying the user's system mailbox, his *mbox* file, or his edit file in −*f*.

from [f] Takes a list of messages and prints their message headers. The default message is neither saved nor deleted.

headers [h] Lists the current range of headers, which is an 18 message group. If the "+" argument is given, then the next 18 message group is printed, and if the "−" argument is given, the previous 18 message group is printed.

help [hel] A synonym for ?

hold [ho] Takes a message list and marks each message therein to be saved in the user's system mailbox instead of in *mbox*. Does not override the **delete** command. The default message must not be deleted.

list [l] The **list** command lists all of the available user commands in the order that the command processor sees them. It takes no arguments.

mail [m] Takes as argument login names and distribution group names and sends mail to those people. Tilde escapes work in **mail**.

next [n] Goes to the next message in sequence and types it. If a message list is given, then **next** goes to the first message in the message list.

preserve [pre] A synonym for **hold**.

print [p] Takes a message list and types out each message on the user's terminal. The default message must not be deleted.

quit [q] Terminates the Mail session, saving all undeleted, unsaved messages in the user's *mbox* file in his login directory, preserving all messages marked with **hold** or **preserve** in his system mailbox, and removing all other messages from his system mailbox. If mail has arrived during the Mail session, the message "You have new mail" is typed. If executing while editing a mailbox file with the −f flag, then the edit file is rewritten. A return to the Shell is effected, unless the rewrite of edit file fails, in which case the user can escape with the **exit** command.

reply [r] Takes a message list and sends mail to each message author just like the **mail** command. The default message must not be deleted.

respond [r] A synonym for **reply**.

save [s] Takes a message list and a filename and appends each message in turn to the end of the file. The filename in quotes, followed by the line count and character count is echoed on the user's terminal. The default message for **save** may not be saved or deleted.

set [se] With no arguments, prints all variable values. Otherwise, sets option. Arguments are of the form "option=value" or "option."

shell [sh] Invokes an interactive version of the shell.

size [si] Takes a message list and prints out the size in characters of each message. The default message for size must not be deleted.

top [to] Takes a message list and prints the top so many lines. The number of lines printed is controlled by the variable "toplines" and defaults to five.

type [t] A synonym for **print**.

unalias [una] Takes a list of names defined by **alias** commands and discards the remembered groups of users. The group names no longer have any significance.

undelete [u] Takes a message list and marks each one as *not* being deleted. Each message in the list must already be deleted. The default message must be deleted.

unset [uns] Takes a list of option names and discards their remembered values; opposite of **set**.

visual [v] Takes a message list and invokes the display editor on each one.

write [w] A synonym for **save**.

xit [x] A synonym for **exit**.

Recall that tilde escapes are used when composing messages to perform special functions. Tilde escapes are only recognized at the beginning of lines. The name "tilde escape" is somewhat of a misnomer since the actual escape character can be set by the option "escape."

Here is a summary of the tilde escapes:

~!command Execute the indicated shell command, then return to the message.

~c name ... Add the given names to the list of carbon copy recipients.

~d Read the file "dead.letter" from your home directory into the message.

~e Invoke the text editor on the message collected so far. After the editing session is finished, you may continue appending text to the message.

~h Edit the message header fields by typing each one in turn and allowing the user to append text to the end or modify the field by using the current terminal erase and kill characters.

~m messages Read the named messages into the message being sent, shifted right one tab. If no messages are specified, read the current message.

~p Print out the message collected so far, prefaced by the message header fields.

~q Abort the message being sent, copying the message to "dead.letter" in your home directory if "save" is set.

~r filename Read the named file into the message.

~s string Cause the named string to become the current subject field.

~t name ... Add the given names to the direct recipient list.

~v Invoke an alternate editor (defined by the VISUAL option) on the message collected so far. Usually, the alternate editor will be a visual editor. After you quit the editor, you may resume appending text to the end of your message.

~w filename Write the message onto the named file.

~|command Pipe the message through the command as a filter. If the command gives no output or terminates abnormally, retain the original text of the message.

~~string Insert the string of text in the message prefaced by a single ~. If you have changed the escape character, then you should double *that* character in order to send it.

Options are controlled via the **set** and **unset** commands. Options may be either binary, in which case it is only significant to see whether they are set or not, or string, in which case it's actual value is of interest.

The binary options include the following:

append Causes messages saved in *mbox* to be appended to the end rather than prepended.

ask 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.

askcc	Causes you to be prompted for additional carbon copy recipients at the end of each message. Responding with a newline indicates your satisfaction with the current list.
autoprint	Causes the **delete** command to behave like **dp** – thus, after deleting a message, the next one will be typed automatically.
ignore	Causes interrupt signals from your terminal to be ignored and echoed as @'s.
metoo	Usually, when a group is expanded that contains the sender, the sender is removed from the expansion. Setting this option causes the sender to be included in the group.
quiet	Suppresses the printing of the version when Mail is first invoked.
save	Causes the message collected prior to a RUBOUT to be saved on the file "dead.letter" in your home directory on receipt of the RUBOUT. Also causes the message to be so saved in the same fashion for ~q.

The following options have string values:

EDITOR	Pathname of the text editor to use in the **edit** command and ~e escape. If not defined, then a default editor is used.
SHELL	Pathname of the shell to use in the ! command and the ~! escape. A default shell is used if this option is not defined.
VISUAL	Pathname of the text editor to use in the **visual** command and ~v escape.
escape	If defined, the first character of this option gives the character to use in the place of ~ to denote escapes.
record	If defined, gives the pathname of the file used to record all outgoing mail. If not defined, then outgoing mail is not so saved.
toplines	If defined, gives the number of lines of a message to be printed out with the **top** command; normally, the first five lines are printed.

8. Conclusion

I would like to acknowledge the suggestions and criticisms of Eric Allman, Ken Arnold, Bob Fabry, Richard Fateman, Bob Kridle, Doug Merritt, David Mosher, Eric Schmidt, Polly Siegel, Michael Ubell, and especially Bill Joy, who sneezed nearby; I caught the bug.

−ME REFERENCE MANUAL

Release 1.1/20

Eric P. Allman

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, pointsizes, 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. In NROFF font 8 is underlined, and is set in bold font in TROFF (although font 3, bold in TROFF, is not underlined in NROFF). Font 0 is no font change; the font of the surrounding text is used instead. Notice that fonts 0 and 8 are "pseudo-fonts"; that is, they are simulated by the macros. This means that although it is legal to set a font register to zero or eight, it is not legal to use the escape character form, such as:

```
\f8
```

All distances are in basic units, so it is nearly always necessary to use a scaling factor. For example, the request to set the paragraph indent to eight one-en spaces is:

```
.nr pi 8n
```

and not

```
.nr pi 8
```

which would set the paragraph indent to eight basic units, or about 0.02 inch. 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 names consisting of letters and digits, with at least one upper case letter. In no case should special characters be used in user-defined names.

On daisy wheel type printers in twelve pitch, the −rv1 flag can be stated to make lines default to one eighth inch (the normal spacing for a newline in twelve-pitch). This is normally

---

†NROFF and TROFF are Trademarks of Bell Laboratories.

too small for easy readability, so the default is to space one sixth inch.

This documentation was TROFF'ed on June 4, 1979 and applies to version 1.1/20 of the -me macros.

1. Paragraphing

These macros are used to begin paragraphs. The standard paragraph macro is .pp; the others are all variants to be used for special purposes.

The first call to one of the paragraphing macros defined in this section or the .sh macro (defined in the next section) *initializes* the macro processor. After initialization it is not possible to use any of the following requests: .sc, .lo, .th, or .ae. Also, the effects of changing parameters which will have a global effect on the format of the page (notably page length and header and footer margins) are not well defined and should be avoided.

.lp — Begin left-justified paragraph. Centering and underlining are turned off if they were on, the font is set to \n(pf [1] the type size is set to \n(pp [10p], and a \n(ps space is inserted before the paragraph [0.35v in TROFF, 1v or 0.5v in NROFF depending on device resolution]. The indent is reset to \n(SI [0] plus \n(po [0] unless the paragraph is inside a display. (see .ba). At least the first two lines of the paragraph are kept together on a page.

.pp — Like .lp, except that it puts \n(pi [5n] units of indent. This is the standard paragraph macro.

.ip *T I* — Indented paragraph with hanging tag. The body of the following paragraph is indented *I* spaces (or \n(ii [5n] spaces if *I* is not specified) more than a non-indented paragraph (such as with .pp) is. The title *T* is exdented (opposite of indented). The result is a paragraph with an even left edge and *T* printed in the margin. Any spaces in *T* must be unpaddable.

.np — A variant of .ip which numbers paragraphs. Numbering is reset after a .lp, .pp, or .sh. The current paragraph number is in \n(Sp.

2. Section Headings

Numbered sections are similar to paragraphs except that a section number is automatically generated for each one. The section numbers are of the form 1.2.3. The *depth* of the section is the count of numbers (separated by decimal points) in the section number.

Unnumbered section headings are similar, except that no number is attached to the heading.

.sh -*N T a b c d e f* — Begin numbered section of depth *N*. If *N* is missing the current depth (maintained in the number register \n(S0) is used. The values of the individual parts of the section number are maintained in \n(S1 through \n(S6. There is a \n(ss [1v] space before the section. *T* is printed as a section title in font \n(sf [3] and size \n(sp [10p]. The "name" of the section may be accessed via *(Sn. If \n(si is non-zero, the base indent is set to \n(si times the section depth, and the section title is exdented. (See .ba.) 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**.

.Sp *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.

.S0 *T B N* — This macro is called automatically after every call to **.Sp**. It is normally undefined, but may be used to automatically put every section title into the table of contents or for some similiar 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.

.S1 − **.S6** — Traps called just before printing that depth section. May be defined to (for example) give variable spacing before sections. These macros are called from **.Sp**, 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.

.hx — Suppress headers and footers on the next page.

.m1 ±*N* — Set the space between the top of the page and the header [4v].

.m2 ±*N* — Set the space between the header and the first line of text [2v].

.m3 ±*N* — Set the space between the bottom of the text and the footer [2v].

.m4 ±*N* — Set the space between the footer and the bottom of the page [4v].

.ep — End this page, but do not begin the next page. Useful for forcing out footnotes, but other than that hardly every used. Must be followed by a **.bp** or the end of input.

.Sh — Called at every page to print the header. May be redefined to provide fancy (e.g., multi-line) headers, but doing so loses the function of the .he, .fo, .eh, .oh, .ef, and .of requests, as well as the chapter-style title feature of .+c.

.SF — Print footer; same comments apply as in .Sh.

.SH — A normally undefined macro which is called at the top of each page (after outputing the header, initial saved floating keeps, etc.); in other words, this macro is called immediately before printing text on a page. It can be used for column headings and the like.

4. Displays

All displays except centered blocks and block quotes are preceded and followed by an extra \n(bs [same as \n(ps] space. Quote spacing is stored in a separate register; centered blocks have no default initial or trailing space. The vertical spacing of all displays except quotes and centered blocks is stored in register \n(SR instead of \n(Sr.

.D *m f* — Begin list. Lists are single spaced, unfilled text. If *f* is F, the list will be filled. If *m* [D] is I the list is indented by \n(bi [4n]; if M the list is indented to the left margin; if L the list is left justified with respect to the text (different from M only if the base indent (stored in \n(Si and set with .ba) is not zero); and if C the list is centered on a line-by-line basis. The list is set in font \n(df [0]. Must be matched by a .)L. This macro is almost like .b except that no attempt is made to keep the display on one page.

.)l — End list.

.q — Begin major quote. These are single spaced, filled, moved in from the text on both sides by \n(qi [4n], preceded and followed by \n(qs [same as \n(bs] space, and are set in point size \n(qp [one point smaller than surrounding text].

.)q — End major quote.

.b *m f* — Begin block. Blocks are a form of *keep*, where the text of a keep is kept together on one page if possible (keeps are useful for tables and figures which should not be broken over a page). If the block will not fit on the current page a new page is begun, *unless* that would leave more than \n(bt [0] white space at the bottom of the text. If \n(bt is zero, the threshold feature is turned off. Blocks are not filled unless *f* is F, when they are filled. The block will be left-justified if *m* is L, indented by \n(bi [4n] if *m* is I or absent, centered (line-for-line) if *m* is C, and left justified to the margin (not to the base indent) if *m* is M. The block is set in font \n(df [0].

.)b — End block.

.z *m f* — Begin floating keep. Like .b except that the keep is *floated* to the bottom of the page or the top of the next page. Therefore, its position relative to the text changes. The floating keep is preceded and followed by \n(zs [1v] space. Also, it defaults to mode M.

.)z — End floating keep.

.c — Begin centered block. The next keep is centered as a block, rather than on a line-by-line basis as with .b C. This call may be nested inside keeps.

.)c — End centered block.

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(Sd 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 preceeded by \n(fs [0.2v] space, is indented \n(fi [3n] on the first line, and is indented \n(fu [0] from the right margin. Footnotes line up underneath two columned output. If the text of the footnote will not all fit on one page it will be carried over to the next page.
.)f *n*	End footnote. The number register \n(Sf and the associated string ** are incremented if they have been referenced.
.Ss	The macro to output the footnote seperator. 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 preceeded by a \n(xs [0.2v] space. Each entry is "undented" by \n(xu [0.5i]; this register tells how far the page number extends into the right margin.
.)x *P A*	End index entry. The index entry is finished with a row of dots with *A* [null] right justified on the last line (such as for an author's name), followed by *P* [\n%]. If *A* is specified, *P* must be specified; \n% can be used to print the current page number. If *P* is an underscore, no page number and no row of dots are printed.
.xp *x*	Print index *x* [x]. The index is formated in the font, size, and so forth in effect at the time it is printed, rather than at the time it is collected.

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(Ss). The column width, calculated to fill the single column line length with both columns, is stored in \n(Sl. The current column is in \n(Sc. You can test register \n(Sm [1] to see if you are in single column or double column mode. Actually, the request enters *N* [2] columned output.
.1c	Revert to single-column mode.
.bc	Begin column. This is like .bp except that it begins a new column on a new page only if necessary, rather than forcing a whole new page if there is another column left on the current page.

7. Fonts and Sizes

.sz +*P*	The pointsize is set to *P* [10p], and the line spacing is set proportionally. The ratio of line spacing to pointsize is stored in \n(Sr. The ratio used internally by displays and annotations is stored in \n(SR (although this is not used by .sz).
.*x W X*	Set *W* in roman font, appending *X* in the previous font. To append different font requests, use *X* = \c. If no parameters, change to roman font.

.i *W X* — Set *W* in italics, appending *X* in the previous font. If no parameters, change to italic font. Underlines in NROFF.

.b *W X* — Set *W* in bold font and append *X* in the previous font. If no parameters, switch to bold font. In NROFF, underlines.

.rb *W X* — Set *W* in bold font and append *X* in the previous font. If no parameters, switch to bold font. .rb differs from .b in that .rb does not underline in NROFF.

.u *W X* — Underline *W* and append *X*. This is a true underlining, as opposed to the .ul request, which changes to "underline font" (usually italics in TROFF). It won't work right if *W* is spread or broken (including hyphenated). In other words, it is safe in nofill mode only.

.q *W X* — Quote *W* and append *X*. In NROFF this just surrounds *W* with double quote marks (`""`), but in TROFF uses directed quotes.

.bi *W X* — Set *W* in bold italics and append *X*. Actually, sets *W* in italic and overstrikes once. Underlines in NROFF. It won't work right if *W* is spread or broken (including hyphenated). In other words, it is safe in nofill mode only.

.bx *W X* — Sets *W* in a box, with *X* appended. Underlines in NROFF. It won't work right if *W* is spread or broken (including hyphenated). In other words, it is safe in nofill mode only.

8. Roff Support

.ix +*N* — Indent, no break. Equivalent to .in *N*.

.bl *N* — Leave *N* contiguous white space, on the next page if not enough room on this page. Equivalent to a .sp *N* inside a block.

.pa +*N* — Equivalent to .bp.

.ro — Set page number in roman numerals. Equivalent to .af % i

.ar — Set page number in arabic. Equivalent to .af % 1

.nl — Number lines in margin from one on each page.

.n2 *N* — Number lines from *N*, stop if *N* = 0.

.sk — Leave the next output page blank, except for headers and footers. This is used to leave space for a full-page diagram which is produced externally and pasted in later. To get a partial-page paste-in display, say .sv *N*, where *N* is the amount of space to leave; this space will be output immediately if there is room, and will otherwise be output at the top of the next page. However, be warned: if *N* is greater than the amount of available space on an empty page, no space will ever be output.

9. Preprocessor Support

.EQ *m T* — Begin equation. The equation is centered if *m* is C or omitted, indented \n(bi [4n] if *m* is L and left justified if *m* is L. *T* is a tag 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 *x*
: 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 *x* = 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.

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(SI). 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(S0.

.ll +-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-columned output. The current line length is stored in \n(SI.

.hl
: Draws a horizontal line the length of the page. This is useful inside floating keeps to differentiate between the text and the figure.

.lo
: This macro loads another set of macros (in /usr/lib/me/local.me) which is intended to be a set of locally defined macros. These macros should all be of the form .*X, where *X* is any letter (upper or lower case) or digit.

11. Standard Papers

.tp
: Begin title page. Spacing at the top of the page can occur, and headers and footers are supressed. Also, the page number is not incremented for this page.

.th
: Set thesis mode. This defines the modes acceptable for a doctoral dissertation at Berkeley. It double spaces, defines the header to be a single page number, and changes the margins to be 1.5 inch on the left and one inch on the top. .++ and .+c should be used with it. This macro must be stated before initialization, that is, before the first call of a paragraphing macro or .sh.

.++ *m H*
: This request defines the section of the paper which we are entering. The section type is defined by *m*. C means that we are entering the chapter portion of the paper. A means that we are entering the appendix portion of the paper. P means that the material following should be the preliminary portion (abstract, table of contents, etc.) portion of the paper. AB means that we are entering the abstract (numbered independently from 1 in Arabic numerals), and B means that we are entering the bibliographic portion at the end of the paper. Also, the variants RC and RA are allowed, which specify renumbering of pages from one at the beginning of each chapter or appendix, respectively. The *H* parameter defines the new header. If there are any spaces in it, the entire header must be quoted. If you want the header to have the chapter

number in it. Use the string \\\\n(ch. For example, to number appendixes A.1 etc., type .++ RA '"\\\\n(ch.%'. Each section (chapter, appendix, etc.) should be preceded by the .++c request. It should be mentioned that it is easier when using TROFF to put the front material at the end of the paper, so that the table of contents can be collected and output; this material can then be physically moved to the beginning of the paper.

.++c *T* — Begin chapter with title *T*. The chapter number is maintained in \n(ch. This register is incremented every time .++c is called with a parameter. The title and chapter number are printed by .Sc. The header is moved to the footer on the first page of each chapter. If *T* is omitted, .Sc is not called; this is useful for doing your own "title page" at the beginning of papers without a title page proper. .Sc calls .SC as a hook so that chapter titles can be inserted into a table of contents automatically.

.Sc *T* — Print chapter number (from \n(ch) and *T*. This macro can be redefined to your liking. It is defined by default to be acceptable for a PhD thesis at Berkeley. This macro calls SC, which can be defined to make index entries, or whatever.

.SC *K N T* — This macro is called by .Sc. It is normally undefined, but can be used to automatically insert index entries, or whatever. *K* is a keyword, either "Chapter" or "Appendix" (depending on the .++ mode); *N* is the chapter or appendix number, and *T* is the chapter or appendix title.

.ac *A N* — This macro (short for .acm) sets up the NROFF environment for photo-ready papers as used by the ACM. This format is 25% larger, and has no headers or footers. The author's name *A* is printed at the bottom of the page (but off the part which will be printed in the conference proceedings), together with the current page number and the total number of pages *N*. Additionally, this macro loads the file /usr/lib/me/acm.me, which may later be augmented with other macros useful for printing papers for ACM conferences. It should be noted that this macro will not work correctly in TROFF, since it sets the page length wider than the physical width of the phototypesetter roll.

12. Predefined Strings

* — Footnote number, actually *[\n(S#]*. This macro is incremented after each call to .)f.

*# — Delayed text number. Actually [\n(Sd].

*[— Superscript. This string gives upward movement and a change to a smaller point size if possible, otherwise it gives the left bracket character ('[').

*] — 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.

*> — Inverse to *<.

*(dw — The day of the week, as a word.

*(mo — The month, as a word.

*(dy — Today's date, directly printable. The date is of the form June 4, 1979. Other forms of the date can be used by using \n(dy (the day of the month; for example, 4), *(mo (as noted above) or \n(mo (the same, but as an ordinal number; for example, June is 6), and \n(yr (the last two digits of the current year).

*(lq	Left quote marks. Double quote in NROFF.
*(rq	Right quote.
*−	¼ 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	*'	a*'	á
Grave accent	*`	e*`	è
Umlat	*:	u*:	ü
Tilde	*~	n*~	ñ
Caret	*^	e*^	ê
Cedilla	*,	c*,	ç
Czech	*v	e*v	ě
Circle	*o	A*o	Å
There exists	*(qe		∃
For all	*(qa		∀

Acknowledgments

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; and the plethora of people who have contributed ideas and have given support for the project.

WRITING PAPERS WITH NROFF USING −ME

Eric P. Allman

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 −T*type 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

.sp 4

spaces four lines. The number 4 is an *argument* to the .sp request which says to space four lines instead of one. Arguments are separated from the request and from each other by spaces.

---

†UNIX, NROFF, and TROFF are Trademarks of Bell Laboratories

1. Basics of Text Processing

The primary function of NROFF is to *collect* words from input lines, *fill* output lines with those words, *justify* the right hand margin by inserting extra spaces in the line, and output the result. For example, the input:

```
Now is the time
for all good men
to come to the aid
of their party.
Four score and seven
years ago....
```

will be read, packed onto output lines, and justified to produce:

> Now is the time for all good men to come to the aid of their party. Four score and seven years ago....

Sometimes you may want to start a new output line even though the line you are on is not yet full; for example, at the end of a paragraph. To do this you can cause a *break*, which starts a new output line. Some requests cause a break automatically, as do blank input lines and input lines beginning with a space.

Not all input lines are text to be formatted. Some of the input lines are *requests* which describe how to format the text. Requests always have a period or an apostrophe ("'") as the first character of the input line.

The text formatter also does more complex things, such as automatically numbering pages, skipping over page folds, putting footnotes in the correct place, and so forth.

I can offer you a few hints for preparing text for input to NROFF. First, keep the input lines short. Short input lines are easier to edit, and NROFF will pack words onto longer lines for you anyhow. In keeping with this, it is helpful to begin a new line after every period, comma, or phrase, since common corrections are to add or delete sentences or phrases. Second, do not put spaces at the end of lines, since this can sometimes confuse the NROFF processor. Third, do not hyphenate words at the end of lines (except words that should have hyphens in them, such as "mother-in-law"); NROFF is smart enough to hyphenate words for you as needed, but is not smart enough to take hyphens out and join a word back together. Also, words such as "mother-in-law" should not be broken over a line, since then you will get a space where not wanted, such as "mother- in-law".

2. Basic Requests

2.1. Paragraphs

Paragraphs are begun by using the .pp request. For example, the input:

```
.pp
Now is the time for all good men
to come to the aid of their party.
Four score and seven years ago....
```

produces a blank line followed by an indented first line. The result is:

> Now is the time for all good men to come to the aid of their party. Four score and seven years ago....

Notice that the sentences of the paragraphs *must not* begin with a space, since blank lines and lines beginning with spaces cause a break. For example, if I had typed:

```
.pp
Now is the time for all good men
    to come to the aid of their party.
Four score and seven years ago....
```

The output would be:

> Now is the time for all good men
> to come to the aid of their party. Four score and seven years ago....

A new line begins after the word "men" because the second line began with a space character.

There are many fancier types of paragraphs, which will be described later.

2.2. Headers and Footers

Arbitrary headers and footers can be put at the top and bottom of every page. Two requests of the form .he *title* and .fo *title* define the titles to put at the head and the foot of every page, respectively. The titles are called *three-part* titles, that is, there is a left-justified part, a centered part, and a right-justified part. To separate these three parts the first character of *title* (whatever it may be) is used as a delimiter. Any character may be used, but backslash and double quote marks should be avoided. The percent sign is replaced by the current page number whenever found in the title. For example, the input:

```
.he '%'
.fo 'Jane Jones''My Book'
```

results in the page number centered at the top of each page, "Jane Jones" in the lower left corner, and "My Book" in the lower right corner.

2.3. Double Spacing

NROFF will double space output text automatically if you use the request .ls 2, as is done in this section. You can revert to single spaced mode by typing .ls 1.

2.4. Page Layout

A number of requests allow you to change the way the printed copy looks, sometimes called the *layout* of the output page. Most of these requests adjust the placing of "white space" (blank lines or spaces). In these explanations, characters in italics should be replaced with values you wish to use; bold characters represent characters which should actually be typed.

The .bp request starts a new page.

The request .sp *N* leaves *N* lines of blank space. *N* can be omitted (meaning skip a single line) or can be of the form *N*i (for *N* inches) or *N*c (for *N* centimeters). For example, the input:

```
.sp 1.5i
My thoughts on the subject
.sp
```

leaves one and a half inches of space, followed by the line "My thoughts on the subject", followed by a single blank line.

The .in ±*N* request changes the amount of white space on the left of the page (the *indent*). The argument *N* can be of the form +*N* (meaning leave *N* spaces more than you are already leaving), −*N* (meaning leave less than you do now), or just *N* (meaning leave exactly *N* spaces). *N* can be of the form *N*i or *N*c also. For example, the input:

```
initial text
.in 5
some text
.in -1i
more text
.in -2c
final text
```

produces "some text" indented exactly five spaces from the left margin, "more text" indented five spaces plus one inch from the left margin (fifteen spaces on a pica typewriter), and "final text" indented five spaces plus one inch minus two centimeters from the margin. That is, the output is:

```
initial text
          some text
                              more text
               final text
```

The .ti +N (temporary indent) request is used like .in +N when the indent should apply to one line only, after which it should revert to the previous indent. For example, the input:

```
.in 1i
.ti 0
Ware, James R.  The Best of Confucius,
Halcyon House, 1950.
An excellent book containing translations of
most of Confucius' most delightful sayings.
A definite must for anyone interested in the early foundations
of Chinese philosophy.
```

produces:

Ware, James R. The Best of Confucius, Halcyon House, 1950. An excellent book containing translations of most of Confucius' most delightful sayings. A definite must for anyone interested in the early foundations of Chinese philosophy.

Text lines can be centered by using the .ce request. The line after the .ce is centered (horizontally) on the page. To center more than one line, use .ce N (where N is the number of lines to center), followed by the N lines. If you want to center many lines but don't want to count them, type:

```
.ce 1000
lines to center
.ce 0
```

The .ce 0 request tells NROFF to center zero more lines, in other words, stop centering.

All of these requests cause a break; that is, they always start a new line. If you want to start a new line without performing any other action, use .br.

2.5. Underlining

Text can be underlined using the .ul request. The .ul request causes the next input line to be underlined when output. You can underline multiple lines by stating a count of *input* lines to underline, followed by those lines (as with the .ce request). For example, the input:

```
.ul 2
Notice that these two input lines
are underlined.
```

will underline those eight words in NROFF. (In TROFF they will be set in italics.)

3. Displays

Displays are sections of text to be set off from the body of the paper. Major quotes, tables, and figures are types of displays, as are all the examples used in this document. All displays except centered blocks are output single spaced.

3.1. Major Quotes

Major quotes are quotes which are several lines long, and hence are set in from the rest of the text without quote marks around them. These can be generated using the commmands .(q and .)q to surround the quote. For example, the input:

```
As Weizenbaum points out:
.(q
It is said that to explain is to explain away.
This maxim is nowhere so well fulfilled
as in the areas of computer programming....
.)q
```

generates as output:

As Weizenbaum points out:

> It is said that to explain is to explain away. This maxim is nowhere so well fulfilled as in the areas of computer programming....

3.2. Lists

A *list* is an indented, single spaced, unfilled display. Lists should be used when the material to be printed should not be filled and justified like normal text, such as columns of figures or the examples used in this paper. Lists are surrounded by the requests .(l and .)l. For example, type:

```
Alternatives to avoid deadlock are:
.(l
Lock in a specified order
Detect deadlock and back out one process
Lock all resources needed before proceeding
.)l
```

will produce:

Alternatives to avoid deadlock are:

```
Lock in a specified order
Detect deadlock and back out one process
Lock all resources needed before proceeding
```

3.3. Keeps

A *keep* is a display of lines which are kept on a single page if possible. An example of where you would use a keep might be a diagram. Keeps differ from lists in that lists may be broken over a page boundary whereas keeps will not.

Blocks are the basic kind of keep. They begin with the request .(b and end with the request .)b. If there is not room on the current page for everything in the block, a new page is begun. This has the unpleasant effect of leaving blank space at the bottom of the page. When this is not appropriate, you can use the alternative, called *floating keeps*.

Floating keeps move relative to the text. Hence, they are good for things which will be referred to by name, such as "See figure 3". A floating keep will appear at the bottom of the current page if it will fit; otherwise, it will appear at the top of the next page. Floating keeps begin with the line .(z and end with the line .)z. For an example of a

floating keep, see figure 1. The .bi request is used to draw a horizontal line so that the figure stands out from the text.

3.4. Fancier Displays

Keeps and lists are normally collected in *nofill* mode, so that they are good for tables and such. If you want a display in fill mode (for text), type .(l F (Throughout this section, comments applied to .(l also apply to .(b and .(z). This kind of display will be indented from both margins. For example, the input:

```
.(l F
And now boys and girls,
a newer, bigger, better toy than ever before!
Be the first on your block to have your own computer!
Yes kids, you too can have one of these modern
data processing devices.
You too can produce beautifully formatted papers
without even batting an eye!
.)l
```

will be output as:

> And now boys and girls, a newer, bigger, better toy than ever before! Be the first on your block to have your own computer! Yes kids, you too can have one of these modern data processing devices. You too can produce beautifully formatted papers without even batting an eye!

Lists and blocks are also normally indented (floating keeps are normally left justified). To get a left-justified list, type .(l L. To get a list centered line-for-line, type .(l C. For example, to get a filled, left justified list, enter:

```
.(l L F
text of block
.)l
```

The input:

```
.(l
first line of unfilled display
more lines
.)l
```

produces the indented text:

> .(z
> .bi
> Text of keep to be floated.
> .sp
> .ce
> Figure 1. Example of a Floating Keep.
> .bi
> .)z
>
> Figure 1. Example of a Floating Keep.

```
first line of unfilled display
more lines
```

Typing the character L after the .(l request produces the left justified result:

```
first line of unfilled display
more lines
```

Using C instead of L produces the line-at-a-time centered output:

```
first line of unfilled display
more lines
```

Sometimes it may be that you want to center several lines as a group, rather than centering them one line at a time. To do this use centered blocks, which are surrounded by the requests .(c and .)c. All the lines are centered as a unit, such that the longest line is centered and the rest are lined up around that line. Notice that lines do not move relative to each other using centered blocks, whereas they do using the C argument to keeps.

Centered blocks are *not* keeps, and may be used in conjunction with keeps. For example, to center a group of lines as a unit and keep them on one page, use:

```
.(b L
.(c
first line of unfilled display
more lines
.)c
.)b
```

to produce:

```
first line of unfilled display
more lines
```

If the block requests (.(b and .)b) had been omitted the result would have been the same, but with no guarantee that the lines of the centered block would have all been on one page. Note the use of the L argument to .(b; this causes the centered block to center within the entire line rather than within the line minus the indent. Also, the center requests must be nested *inside* the keep requests.

4. Annotations

There are a number of requests to save text for later printing. *Footnotes* are printed at the bottom of the current page. *Delayed text* is intended to be a variant form of footnote: the text is printed only when explicitly called for, such as at the end of each chapter. *Indexes* are a type of delayed text having a tag (usually the page number) attached to each entry after a row of dots. Indexes are also saved until called for explicitly.

4.1. Footnotes

Footnotes begin with the request .(f and end with the request .)f. The current footnote number is maintained automatically, and can be used by typing **, to produce a footnote number[1]. The number is automatically incremented after every footnote. For example, the input:

[1] Like this.

```
.(q
A man who is not upright
and at the same time is presumptuous:
one who is not diligent and at the same time is ignorant:
one who is untruthful and at the same time is incompetent:
such men I do not count among acquaintances.\**
.(f
\**James R. Ware,
.ul
The Best of Confucius,
Halcyon House, 1950.
Page 77.
.)f
.)q
```

generates the result:

> A man who is not upright and at the same time is presumptuous: one who is not diligent and at the same time is ignorant: one who is untruthful and at the same time is incompetent: such men I do not count among acquaintances.[1]

It is important that the footnote appears *inside* the quote, so that you can be sure that the footnote will appear on the same page as the quote.

4.2. Delayed Text

Delayed text is very similar to a footnote except that it is printed when called for explicitly. This allows a list of references to appear (for example) at the end of each chapter, as is the convention in some disciplines. Use *# on delayed text instead of ** as on footnotes.

If you are using delayed text as your standard reference mechanism, you can still use footnotes, except that you may want to reference them with special characters* rather than numbers.

4.3. Indexes

An "index" (actually more like a table of contents, since the entries are not sorted alphabetically) resembles delayed text, in that it is saved until called for. However, each entry has the page number (or some other tag) appended to the last line of the index entry after a row of dots.

Index entries begin with the request .(x and end with .)x. The .)x request may have a argument, which is the value to print as the "page number". It defaults to the current page number. If the page number given is an underscore ("_") no page number or line of dots is printed at all. To get the line of dots without a page number, type .)x "", which specifies an explicitly null page number.

The .xp request prints the index.

For example, the input

---

[1] James R. Ware, *The Best of Confucius*, Halcyon House, 1950, Page 77.

*Such as an asterisk.

```
.(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 "indicies" may be maintained simultaneously (such as a list of tables, table of contents, etc.).

Notice that the index must be printed at the *end* of the paper, rather than at the beginning where it will probably appear (as a table of contents); the pages may have to be physically rearranged after printing.

5. Fancier Features

A large number of fancier requests exist, notably requests to provide other sorts of paragraphs, numbered sections of the form 1.2.3 (such as used in this document), and multicolumn output.

5.1. More Paragraphs

Paragraphs generally start with a blank line and with the first line indented. It is possible to get left-justified block-style paragraphs by using .lp instead of .pp, as demonstrated by the next paragraph.

Sometimes you want to use paragraphs that have the *body* indented, and the first line exdented (opposite of indented) with a label. This can be done with the .ip request. A word specified on the same line as .ip is printed in the margin, and the body is lined up at a prespecified position (normally five spaces). For example, the input:

```
.ip one
This is the first paragraph.
Notice how the first line
of the resulting paragraph lines up
with the other lines in the paragraph.
.ip two
And here we are at the second paragraph already.
You may notice that the argument to .ip
appears
in the margin.
.lp
We can continue text...
```

produces as output

one This is the first paragraph. Notice how the first line of the resulting paragraph lines up with the other lines in the paragraph.

two And here we are at the second paragraph already. You may notice that the argument to .ip appears in the margin.

We can continue text without starting a new indented paragraph by using the .lp request.

If you have spaces in the label of a .ip request, you must use an "unpaddable space" instead of a regular space. This is typed as a backslash character ("\") followed by a space. For example, to print the label "Part 1", enter:

```
.ip "Part\ 1"
```

If a label of an indented paragraph (that is, the argument to .ip) is longer than the space allocated for the label, the label will not be separated from the text, and the rest of the text will be lined up at the old margin (and not with the first line of text). For example, the input:

```
.ip longlabel
This paragraph had a long label.
The first character of text on the first line
will not line up with the text on second and subsequent lines,
although they will line up with each other.
```

will produce:

longlaThis paragraph had a long label. The first character of text on the first line will not line up with the text on second and subsequent lines, although they will line up with each other.

It is possible to change the size of the label by using a second argument which is the size of the label. For example, the above example could be done correctly by saying:

```
.ip longlabel 10
```

which will make the paragraph indent 10 spaces for this paragraph only. If you have many paragraphs to indent all the same amount, use the *number register* ii. For example, to leave one inch of space for the label, type:

```
.nr ii 1i
```

somewhere before the first call to .ip. Refer to the reference manual for more information.

If .ip is used with no argument at all no hanging tag will be printed. For example, the input:

```
.ip [a]
This is the first paragraph of the example.
We have seen this sort of example before.
.ip
This paragraph is lined up with the previous paragraph,
but it has no tag in the margin.
```

produces as output:

[a] This is the first paragraph of the example. We have seen this sort of example before.

This paragraph is lined up with the previous paragraph, but it has no tag in the margin.

A special case of .ip is .np, which automatically numbers paragraphs sequentially from 1. The numbering is reset at the next .pp, .lp, or .sh (to be described in the next section) request. For example, the input:

```
.np
This is the first point.
.np
This is the second point.
Points are just regular paragraphs
which are given sequence numbers automatically
by the .np request.
.pp
This paragraph will reset numbering by .np.
.np
For example,
we have reverted to numbering from one now.
```

generates:

(1) This is the first point.

(2) This is the second point. Points are just regular paragraphs which are given sequence numbers automatically by the .np request.

This paragraph will reset numbering by .np.

(1) For example, we have reverted to numbering from one now.

5.2. Section Headings

Section numbers (such as the ones used in this document) can be automatically generated using the .sh request. You must tell .sh the *depth* of the section number and a section title. The depth specifies how many numbers are to appear (separated by decimal points) in the section number. For example, the section number 4.2.5 has a depth of three.

Section numbers are incremented in a fairly intuitive fashion. If you add a number (increase the depth), the new number starts out at one. If you subtract section numbers (or keep the same number) the final number is incremented. For example, the input:

```
.sh 1 "The Preprocessor"
.sh 2 "Basic Concepts"
.sh 2 "Control Inputs"
.sh 3
.sh 3
.sh 1 "Code Generation"
.sh 3
```

produces as output the result:

1. The Preprocessor
1.1. Basic Concepts
1.2. Control Inputs
1.2.1.
1.2.2.
2. Code Generation
2.1.1.

You can specify the section number to begin by placing the section number after the section title, using spaces instead of dots. For example, the request

.sh 3 "Another section" 7 3 4

will begin the section numbered 7.3.4; all subsequent .sh requests will number relative to this number.

There are more complex features which will cause each section to be indented proportionally to the depth of the section. For example, if you enter:

.nr si N

each section will be indented by an amount N. N must have a scaling factor attached, that is, it must be of the form Nx, where *x* is a character telling what units N is in. Common values for *x* are i for inches, c for centimeters, and n for *ens* (the width of a single character). For example, to indent each section one-half inch, type:

.nr si 0.5i

After this, sections will be indented by one-half inch per level of depth in the section number. For example, this document was produced using the request

.nr si 3n

at the beginning of the input file, giving three spaces of indent per section depth.

Section headers without automatically generated numbers can be done using:

.uh "Title"

which will do a section heading, but will put no number on the section.

5.3. Parts of the Basic Paper

There are some requests which assist in setting up papers. The .tp request initializes for a title page. There are no headers or footers on a title page, and unlike other pages you can space down and leave blank space at the top. For example, a typical title page might appear as:

```
.tp
.sp 2i
.(l C
THE GROWTH OF TOENAILS
IN UPPER PRIMATES
.sp
by
.sp
Frank N. Furter
.)l
.bp
```

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 pretty tables in a variety of formats. This document will only describe the embellishments to the standard features; consult the reference manuals for those processors for a description of their use.

The eqn and neqn programs are described fully in the document *Typesetting Mathematics — Users' Guide* by Brian W. Kernighan and Lorinda L. Cherry. Equations are centered, and are kept on one page. They are introduced by the .EQ request and terminated by the .EN request.

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

```
.th                          \" set for thesis mode
.fo "DRAFT"                  \" define footer for each page
.tp                          \" begin title page
.(l C                        \" center a large block
THE GROWTH OF TOENAILS
IN UPPER PRIMATES
.sp
by
.sp
Frank Furter
.)l                          \" end centered part
.+c INTRODUCTION             \" begin chapter named "INTRODUCTION"
.(x t                        \" make an entry into index 't'
Introduction
.)x                          \" end of index entry
text of chapter one
.+c "NEXT CHAPTER"           \" begin another chapter
.(x t                        \" enter into index 't' again
Next Chapter
.)x
text of chapter two
.+c CONCLUSIONS
.(x t
Conclusions
.)x
text of chapter three
.++ B                        \" begin bibliographic information
.+c BIBLIOGRAPHY             \" begin another 'chapter'
.(x t
Bibliography
.)x
text of bibliography
.++ P                        \" begin preliminary material
.+c "TABLE OF CONTENTS"
.xp t                        \" print index 't' collected above
.+c PREFACE                  \" begin another preliminary section
text of preface
```

Figure 2. Outline of a Sample Paper

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.

5.7. Annotations Inside Keeps

Sometimes you may want to put a footnote or index entry inside a keep. For example, if you want to maintain a "list of figures" you will want to do something like:

```
.(z
.(c
text of figure
.)c
.ce
Figure 5.
.(x f
Figure 5
.)x
.)z
```

which you may hope will give you a figure with a label and an entry in the index f (presumably a list of figures index). Unfortunately, the index entry is read and interpreted when the keep is read, not when it is printed, so the page number in the index is likely to be wrong. The solution is to use the magic string \! at the beginning of all the lines dealing with the index. In other words, you should use:

```
.(z
.(c
Text of figure
.)c
.ce
Figure 5.
\!.(x f
\!Figure 5
\!.)x
.)z
```

which will defer the processing of the index until the figure is output. This will guarantee that the page number in the index is correct. The same comments apply to blocks (with .(b and .)b) as well.

6. TROFF and the Photosetter

With a little care, you can prepare documents that will print nicely on either a regular terminal or when phototypeset using the TROFF formatting program.

6.1. Fonts

A *font* is a style of type. There are three fonts that are available simultaneously, Times Roman, Times Italic, and Times Bold, plus the special math font. The normal font is Roman. Text which would be underlined in NROFF with the .ul request is set in italics in TROFF.

There are ways of switching between fonts. The requests .r, .i, and .b switch to Roman, italic, and bold fonts respectively. You can set a single word in some font by typing (for example):

```
.i word
```

which will set *word* in italics but does not affect the surrounding text. In NROFF, italic and bold text is underlined.

Notice that if you are setting more than one word in whatever font, you must surround that word with double quote marks (`"`) so that it will appear to the NROFF processor as a single word. The quote marks will not appear in the formatted text. If you do want a quote mark to appear, 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 "I" 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 inbetween 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.

6.2. Point Sizes

The phototypesetter supports different sizes of type, measured in points. The default point size is 10 points for most text, 8 points for footnotes. To change the pointsize, type:

.sz +N

where .N is the size wanted in points. The *vertical spacing* (distance between the bottom of most letters (the *baseline*) between adjacent lines) is set to be proportional to the type size.

Warning: changing point sizes on the phototypesetter is a slow mechanical operation. Size changes should be considered carefully.

6.3. Quotes

It is conventional when using the typesetter to use pairs of grave and acute accents to generate double quotes, rather than the double quote character ('"'). This is because it looks better to use grave and acute accents: for example, compare "quote" to "quote".

In order to make quotes compatible between the typesetter and terminals, you may use the sequences *(lq and *(rq to stand for the left and right quote respectively.

These both appear as " on most terminals, but are typeset as “ and ” respectively. For example, use:

```
\*(lqSome things aren't true
even if they did happen.\*(rq
```

to generate the result:

"Some things aren't true even if they did happen."

As a shorthand, the special font request

```
.q "quoted text"
```

will generate "quoted text". Notice that you must surround the material to be quoted with double quote marks if it is more than one word.

Acknowledgments

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; and the plethora of people who have contributed ideas and have given support for the project.

This document was TROFF'ed on April 29, 1979 and applies to version 1.1 of the -me macros.

Screen Updating and Cursor Movement Optimization: A Library Package

Kenneth C. R. C. Arnold

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

ABSTRACT

This document describes a package of C library functions which allow the user to:

- update a screen with reasonable optimization,
- get input from the terminal in a screen-oriented fashion, and
- independent from the above, move the cursor optimally from one point to another.

These routines all use the **/etc/termcap** database to describe the capabilities of the terminal.

Acknowledgements

This package would not exist without the work of Bill Joy, who, in writing his editor, created the capability to generally describe terminals, wrote the routines which read this database, and, most importantly, those which implement optimal cursor movement, which routines I have simply lifted nearly intact. Doug Merritt and Kurt Shoens also were extremely important, as were both willing to waste time listening to me rant and rave. The help and/or support of Ken Abrams, Alan Char, Joe Kalash, and Bob Kridle, was, and is, also greatly appreciated.

Contents

Appendixes

1. Overview

In making available the generalized terminal descriptions in /etc/termcap, much information was made available to the programmer, but little work was taken out of one's hands. The purpose of this package is to allow the C programmer to do the most common type of terminal dependent functions, those of movement optimization and optimal screen updating, without doing any of the dirty work, and (hopefully) with nearly as much ease as is necessary to simply print or read things.

The package is split into three parts: (1) Screen updating; (2) Screen updating with user input; and (3) Cursor motion optimization.

It is possible to use the motion optimization without using either of the other two, and screen updating and input can be done without any programmer knowledge of the motion optimization, or indeed the database itself.

1.1. Terminology (or, Words You Can Say to Sound Brilliant)

In this document, the following terminology is kept to with reasonable consistency:

window: An internal representation containing an image of what a section of the terminal screen may look like at some point in time. This subsection can either encompass the entire terminal screen, or any smaller portion down to a single character within that screen.

terminal: Sometimes called ***terminal screen***. The package's idea of what the terminal's screen currently looks like, i.e., what the user sees now. This is a special ***screen***.

screen: This is a subset of windows which are as large as the terminal screen, i.e., they start at the upper left hand corner and encompass the lower right hand corner. One of these, *stdscr*, is automatically provided for the programmer.

1.2. Compiling Things

In order to use the library, it is necessary to have certain types and variables defined. Therefore, the programmer must have a line:

#include <curses.h>

at the top of the program source[1]. Also, compilations should have the following form:

cc [*flags*] file ... -lcurses -ltermlib

1.3. Screen Updating

In order to update the screen optimally, it is necessary for the routines to know what the screen currently looks like and what the programmer wants it to look like next. For this purpose, a data type (structure) named *WINDOW* is defined which describes a window image to the routines, including its starting position on the screen (the (y, x) co-ordinates of the upper left hand corner) and its size. One of these (called *curscr* for *current screen*) is a screen image of what the terminal currently looks like. Another screen (called *stdscr*, for *standard screen*) is provided by default to make changes on.

---

[1] The screen package uses the Standard I/O library, so the header file curses.h has the line #in-

A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the terminal screen. It is more like an array of characters on which to make changes.

When one has a window which describes what some part the terminal should look like, the routine *refresh()* (or *wrefresh()* if the window is not *stdscr*) is called. *Refresh()* makes the terminal, in the area covered by the window, look like that window. Note, therefore, that changing something on a window *does not change the terminal*. Actual updates to the terminal screen are made only by calling *refresh()* or *wrefresh()*. This allows the programmer to maintain several different ideas of what a portion of the terminal screen should look like. Also, changes can be made to windows in any order, without regard to motion efficiency. Then, at will, the programmer can effectively say "make it look like this," and let the package worry about the best way to do this.

1.4. Naming Conventions

As hinted above, the routines can use several windows, but two are automatically given: *curscr*, which knows what the terminal looks like, and *stdscr*, which is what the programmer wants the terminal to look like next. The user should never really access *curscr* directly. Changes should be made to the appropriate screen, and then the routine *refresh()* (or *wrefresh()*) should be called.

Many functions are set up to deal with *stdscr* as a default screen. For example, to add a character to *stdscr*, one calls *addch()* with the desired character. If a different window is to be used, the routine *waddch()* (for window-specific *addch()*) is provided[2]. This convention of prepending function names with a "w" when they are to be applied to specific windows is consistent. The only routines which do *not* do this are those to which a window must always be specified.

In order to move the current (y, x) co-ordinates from one point to another, the routines *move()* and *wmove()* are provided. However, it is often desirable to first move and then perform some I/O operation. In order to avoid clumsyness, most I/O routines can be preceded by the prefix "mv" and the desired (y, x) co-ordinates then can be added to the arguments to the function. For example, the calls

```
move(y, x);
addch(ch);
```

can be replaced by

```
mvaddch(y, x, ch);
```

and

```
wmove(win, y, x);
waddch(win, ch);
```

can be replaced by

```
mvwaddch(win, y, x, ch);
```

Note that the window description pointer (*win*) comes before the added (y, x) co-ordinates. If such pointers are need, they are always the first parameters passed.

---

clude <stdio.h> in it. It is therefore redundant (but harmless) for the programmer to do it, too.

[2] Actually, *addch()* is really a "#define" macro with arguments, as are most of the "functions" which deal with *stdscr* as a default.

2. Variables

Many variables which are used to describe the terminal environment are available to the programmer. They are:

type	name	description
WINDOW *	curscr	current version of the screen (terminal screen).
WINDOW *	stdscr	standard screen. Most updates are usually done here.
char *	Def_term	default terminal type if type cannot be determined
bool	My_term	use the terminal specification in *Def_term* as terminal, irrelevant of real terminal type
char *	ttytype	full name of the current terminal.
int	LINES	number of lines on the terminal
int	COLS	number of columns on the terminal
int	ERR	error flag returned by routines on a fail.
int	OK	error flag returned by routines when things go right.

There are also several "#define" constants and types which are of general usefulness:

reg	storage class "register" (e.g., *reg int i;*)
bool	boolean type, actually a "char" (e.g., *bool doneit;*)
TRUE	boolean "true" flag (1).
FALSE	boolean "false" flag (0).

3. Usage

This is a description of how to actually use the screen package. In it, we assume all updating, reading, etc. is applied to *stdscr*. All instructions will work on any window, with changing the function name and parameters as mentioned above.

3.1. Starting up

In order to use the screen package, the routines must know about terminal characteristics, and the space for *curscr* and *stdscr* must be allocated. These functions are performed by *initscr()*. Since it must allocate space for the windows, it can overflow core when attempting to do so. On this rather rare occasion, *initscr()* returns ERR. *Initscr()* must *always* be called before any of the routines which affect windows are used. If it is not, the program will core dump as soon as either *curscr* or *stdscr* are referenced. However, it is usually best to wait to call it until after you are sure you will need it, like after checking for startup errors. Terminal status changing routines like *nl()* and *crmode()* should also be called after *initscr()*.

Now that the screen windows have been allocated, you can set them up for the run. If you want to, say, allow the window to scroll, use *scrollok()*. If you want the cursor to be left after the last change, use *leaveok()*. If this isn't done, *refresh()* will move the cursor to the window's current (y, x) co-ordinates after updating it. New windows of your own can be created, too, by using the functions *newwin()* and *subwin()*. *Delwin()* will allow you to get rid of old windows. If you wish to change the official size of the terminal by hand, just set the variables *LINES* and *COLS* to be what you want, and then call *initscr()*. This is best done before, but can be done either before or after, the first call to *initscr()*, as it will always delete any existing *stdscr* and/or *curscr* before creating new ones.

3.2. The Nitty-Gritty

3.2.1. Output

Now that we have set things up, we will want to actually update the terminal. The basic functions used to change what will go on a window are *addch()* and *move()*. *Addch()* adds a character at the current (y, x) co-ordinates, returning ERR if it would cause the window to illegally scroll, i.e., printing a character in the lower right-hand corner of a terminal which automatically scrolls if scrolling is not allowed. *Move()* changes the current (y, x) co-ordinates to whatever you want them to be. It returns ERR if you try to move off the window when scrolling is not allowed. As mentioned above, you can combine the two into *mvaddch()* to do both things in one fell swoop.

The other output functions, such as *addstr()* and *printw()*, all call *addch()* to add characters to the window.

After you have put on the window what you want there, when you want the portion of the terminal covered by the window to be made to look like it, you must call *refresh()*. In order to optimize finding changes, *refresh()* assumes that any part of the window not changed since the last *refresh()* of that window has not been changed on the terminal, i.e., that you have not refreshed a portion of the terminal with an overlapping window. If this is not the case, the routine *touchwin()* is provided to make it look like the entire window has been changed, thus making *refresh()* check the whole subsection of the terminal for changes.

3.2.2. Input

Input is essentially a mirror image of output. The complementary function to *addch()* is *getch()* which, if echo is set, will call *addch()* to echo the character. Since the screen package needs to know what is on the terminal at all times, if characters are to be echoed, the tty must be in raw or cbreak mode. If it is not, *getch()* sets it to be raw, and then reads in the character.

3.2.3. Miscellaneous

All sorts of fun functions exists for maintaining and changing information about the windows. For the most part, the descriptions in section 5.4. should suffice.

3.3. Finishing up

In order to do certain optimizations, and, on some terminals, to work at all, some things must be done before the screen routines start up. These functions are performed in *gettmode()* and *setterm()*, which are called by *initscr()*. In order to clean up after the routines, the routine *endwin()* is provided. It restores tty modes to what they were when *initscr()* was first called. Thus, anytime after the call to initscr, *endwin()* should be called before exiting.

4. Cursor Motion Optimization: Standing Alone

It is possible to use the cursor optimization functions of this screen package without the overhead and additional size of the screen updating functions. The screen updating functions are designed for uses where parts of the screen are changed, but the overall image remains the same. This includes such programs as eye and vi[3]. Certain other programs will find it considerably difficult to use these functions in this manner without considerable unnecessary pro-

[3] Eye actually uses these functions, vi does not.

gram overhead. For such applications, such as some "*crt hacks*"[4] and optimizing cat(1)-type programs, all that is needed is the motion optimizations. This, therefore, is a description of what some of what goes on at the lower levels of this screen package. The descriptions assume a certain amount of familiarity with programming problems and some finer points of C. None of it is terribly difficult, but you should be forewarned.

4.1. Terminal Information

In order to use a terminal's features to the best of a program's abilities, it must first know what they are[5]. The **/etc/termcap** database describes these, but a certain amount of decoding is necessary, and there are, of course, both efficient and inefficient ways of reading them in. The algorithm that the uses is taken from **vi** and is hideously efficient. It reads them in a tight loop into a set of variables whose names are two uppercase letters with some mnemonic value. For example, *HO* is a string which moves the cursor to the "home" position[6]. As there are two types of variables involving ttys, there are two routines. The first, *gettmode()*, sets some variables based upon the tty modes accessed by gtty(2) and stty(2). The second, *setterm()*, a larger task by reading in the descriptions from the **/etc/termcap** database. This is the way these routines are used by *initscr()*:

```
if (isatty(0)) {
    gettmode();
    if (sp=getenv("TERM"))
        setterm(sp);
}
else
    setterm(Def_term);
_puts(TI);
_puts(VS);
```

Isatty() checks to see if file descriptor 0 is a terminal[7]. If it is, *gettmode()* sets the terminal description modes from a gtty(2). *Getenv()* is then called to get the name of the terminal, and that value (if there is one) is passed to *setterm()*, which reads in the variables from **/etc/termcap** associated with that terminal. (*Getenv()* returns a pointer to a string containing the name of the terminal, which we save in the character pointer *sp*.) If *isatty()* returns false, the default terminal *Def_term* is used. The *TI* and *VS* sequences initialize the terminal (*_puts()* is a macro which uses *tputs()* (see termcap(3)) to put out a string).

4.2. Movement Optimizations, or, Getting Over Yonder

Now that we have all this useful information, it would be nice to do some-

---

[4] Graphics programs designed to run on character-oriented terminals. I could name many, but they come and go, so the list would be quickly out of date. Recently, there have been programs such as rocket and gun.

[5] If this comes as any surprise to you, there's this tower in Paris they're thinking of junking that I can let you have for a song.

[6] These names are identical to those variables used in the /etc/termcap database to describe each capability. See Appendix A for a complete list of those read, and termcap(5) for a full description.

[7] *Isatty()* is defined in the default C library function routines. It does a gtty(2) on the descriptor and checks the return value.

thing with it[8]. The most difficult thing to do properly is motion optimization. When you consider how many different features different terminals have (tabs, backtabs, non-destructive space, home sequences, absolute tabs, ...) you can see that deciding how to get from here to there can be a decidedly non-trivial task. The editor vi uses many of these features, and the routines it uses to do this take up many pages of code. Fortunately, I was able to liberate them with the author's permission, and use them here.

After using *gettmode()* and *setterm()* to get the terminal descriptions, the function *mvcur()* deals with this task. It usage is simple: you simply tell it where you are now and where you want to go. For example

```
mvcur(0, 0, LINES/2, COLS/2)
```

would move the cursor from the home position (0, 0) to the middle of the screen. If you wish to force absolute addressing, you can use the function *tgoto()* from the termlib(7) routines, or you can tell *mvcur()* that you are impossibly far away, like Cleveland. For example, to absolutely address the lower left hand corner of the screen from anywhere just claim that you are in the upper right hand corner:

```
mvcur(0, COLS-1, LINES-1, 0)
```

5. The Functions

In the following definitions, "†" means that the "function" is really a "#define" macro with arguments. This means that it will not show up in stack traces in the debugger, or, in the case of such functions as *addch()*, it will show up as it's "w" counterpart. The arguments are given to show the order and type of each. Their names are not mandatory, just suggestive.

5.1. Output Functions

```
addch(ch) †
char    ch;

waddch(win, ch)
WINDOW  *win;
char    ch;
```

Add the character *ch* on the window at the current (y, x) co-ordinates. This returns ERR if it would cause the screen to scroll illegally.

```
addstr(str) †
char    *str;

waddstr(win, str)
WINDOW  *win;
char    *str;
```

Add the string pointed to by *str* on the window at the current (y, x) co-ordinates. This returns ERR if it would cause the screen to scroll illegally. In this case, it will put on as much as it can.

---

[8] Actually, it can be emotionally fulfilling just to get the information. This is usually only true, however, if you have the social life of a kumquat.

box(win, vert, hor)
*WINDOW *win;*
char vert, hor;

Draws a box around the window using *vert* as the character for drawing the vertical sides, and *hor* for drawing the horizontal lines. If scrolling is not allowed, and the window encompasses the lower right-hand corner of the terminal, the corners are left blank to avoid a scroll.

clear() †

wclear(win)
*WINDOW *win;*

Resets the entire window to blanks. If *win* is a screen, this sets the clear flag, which will cause a clear-screen sequence to be sent on the next *refresh()* call.

clearok(scr, boolf) †
*WINDOW *scr;*
bool boolf;

Sets the clear flag for the screen *scr*. If *boolf* is TRUE, this will force a clear-screen to be printed on the next *refresh()* or *wrefresh()*, or stop it from doing so if *boolf* is FALSE. This only works on screens, and, unlike *clear()*, does not alter the contents of the screen. If *scr* is *curscr*, the next *refresh()* call will cause a clear-screen, even if the window passed to *refresh()* is not a screen.

clrtobot() †

wclrtobot(win)
*WINDOW *win;*

Wipes the window clear from the current (y, x) co-ordinates to the bottom. This does not force a clear-screen sequence on the next refresh under any circumstances. This has no associated "mv" command.

clrtoeol() †

wclrtoeol(win)
*WINDOW *win;*

Wipes the window clear from the current (y, x) co-ordinates to the end of the line. This has no associated "mv" command.

erase() †

werase(win)
*WINDOW *win;*

Erases the window to blanks without setting the clear flag. This is analagous to *clear()*, except that it never causes a clear-screen sequence to be generated on a *refresh()*. This has no associated "mv" command.

move(y, x) †
int y, x;

wmove(win, y, x)
*WINDOW *win;*
int y, x;

Change the current (y, x) co-ordinates of the window to (*y*, *x*). This returns ERR if it would cause the screen to scroll illegally.

overlay(win1, win2)
*WINDOW *win1, *win2;*

Overlay *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done non-destructively, i.e., blanks on *win1* leave the contents of the space on *win2* untouched.

overwrite(win1, win2)
*WINDOW *win1, *win2;*

Overwrite *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done destructively, i.e., blanks on *win1* become blank on *win2*.

printw(fmt, arg1, arg2, ...)
*char *fmt;*

wprintw(win, fmt, arg1, arg2, ...)
*WINDOW *win;*
*char *fmt;*

Performs a *printf()* on the window starting at the current (y, x) co-ordinates. It uses *addstr()* to add the string on the window. It is often advisable to use the field width options of *printf()* to avoid leaving things on the window from earlier calls. This returns ERR if it would cause the screen to scroll illegally.

refresh() †

wrefresh(win)
*WINDOW *win;*

Synchronize the terminal screen with the desired window. If the window is not a screen, only that part covered by it is updated. This returns ERR if it would cause the screen to scroll illegally. In this case, it will update whatever it can without causing the scroll.

standout() †

wstandout(win)
*WINDOW *win;*

standend() †

wstandend(win)
*WINDOW *win;*

Start and stop putting characters in standout mode *standout()* causes any characters added to the window to be put in standout mode on the terminal (if it has that capability). *standend()* stops this. The sequences *SO* and *SE* (or *US* and *UE* if they are not defined) are used (see Appendix A).

5.2. Input Functions

crmode() †

nocrmode() †

Set or unset the terminal to/from cbreak mode.

echo() †

noecho() †

Sets the terminal to echo or not echo characters.

getch() †

wgetch(win)
*WINDOW *win;*

Gets a character from the terminal and (if necessary) echos it on the window. This returns ERR if it would cause the screen to scroll illegally. Otherwise, the character gotten is returned. If *noecho* has been set, then the window is left unaltered. In order to retain control of the terminal, it is necessary to have one of *noecho*, *cbreak*, or *rawmode* set. If you do not set one, whatever routine you call to read characters will set *cbreak* for you, and then reset to the original mode when finished.

getstr(str) †
*char *str;*

wgetstr(win, str)
*WINDOW *win;*
*char *str;*

Get a string through the window and put it in the location pointed to by *str*, which is assumed to be large enough to handle it. It sets tty modes if necessary, and then calls *getch()* (or *wgetch(win)*) to get the characters needed to fill in the string until a newline or EOF is encountered. This returns ERR if it would cause the screen to scroll illegally.

raw() †

noraw() †

Set or unset the terminal to/from raw mode. On version 7 UNIX® this also turns of newline mapping (see *nl()*).

® UNIX is a trademark of Bell Laboratories.

scanw(fmt, arg1, arg2, ...)
char **fmt;*

wscanw(win, fmt, arg1, arg2, ...)
WINDOW **win;*
char **fmt;*

Perform a *scanf()* through the window using *fmt*. It does this using consecutive *getch()*'s (or *wgetch(win)*'s). This returns ERR if it would cause the screen to scroll illegally.

5.3. Miscellaneous Functions

delwin(win)
WINDOW **win;*

Deletes the window from existence. All resources are freed for future use by *calloc()*, *cfree()*, etc. If a window has a *subwin()* allocated window inside of it, deleting the outer window the subwindow is not affected, even though this does invalidate it. Therefore, subwindows should be deleted before their outer windows are.

endwin()

Finish up window routines before exit. This restores the terminal to the state it was before *initscr()* (or *gettmode()* and *setterm()*) was called. It should always be called before exiting. It does not exit. This is especially useful for resetting tty stats when trapping rubouts via *signal()*.

getyx(win, y, x) †
WINDOW **win;*
int *y, x;*

Puts the current (y, x) co-ordinates of *win* in the variables *y* and *x*. Since it is a macro, not a function, you do not pass the address of *y* and *x*.

inch() †

winch(win) †
WINDOW **win;*

Returns the character at the current (y, x) co-ordinates on the given window. This does not make any changes to the window. This has no associated "**mv**" command.

initscr()

Initialize the screen routines. This must be called before any of the screen routines are used. It initializes the terminal-type data and such, and without it, none of the routines can operate. If standard input is not a tty, it sets the specifications to the terminal whose name is pointed to by *Def_term* (initialy "dumb"). If the boolean *My_term* is true, *Def_term* is always used.

leaveok(win, boolf) †
WINDOW **win;*
bool *boolf;*

Sets the boolean flag for leaving the cursor after the last change. If *boolf* is TRUE, the cursor will be left after the last update on the terminal, and the current (y, x) co-ordinates for *win* will be changed accordingly. If it is FALSE, it will be moved to the current (y, x) co-ordinates. This flag (initialy FALSE) retains its value until changed by the user.

longname(termbuf, name)
char **termbuf, *name;*

Fills in *name* with the long (full) name of the terminal described by the termcap entry in *termbuf*. It is generally of little use, but is nice for telling the user in a readable format what terminal we think he has. This is available in the global variable *ttytype*. *Termbuf* is usually set via the termlib routine *tgetent()*.

mvwin(win, y, x)
WINDOW **win;*
int *y, x;*

Move the position of the window *win* from its current starting coordinates to (*y, x()*). If it does not fit at that position, *mvwin()* returns ERR and does not change anything

WINDOW ***newwin(lines, cols, begin_y, begin_x)**
int *lines, cols, begin_y, begin_x;*

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y, begin_x*). If either *lines* or *cols* is 0 (zero), that dimension will be set to (*LINES* − *begin_y*) or (*COLS* − *begin_x*) respectively. Thus, to get a new window of dimensions *LINES* × *COLS*, use *newwin(0, 0, 0, 0)*.

nl() †

nonl() †

Set or unset the terminal to/from nl mode, i.e., start/stop the system from mapping <RETURN> to <LINE-FEED>. If the mapping is not done, *refresh()* can do more optimization, so it is recommended, but not required, to turn it off.

scroll(win)
WINDOW **win;*

Scroll the window upward one line. This is normally not used by the user.

scrollok(win, boolf) †
WINDOW **win;*
bool *boolf;*

Set the scroll flag for the given window. If *boolf* is FALSE, scrolling is not allowed. This is its default setting.

touchwin(win)
WINDOW **win;*

Make it appear that the every location on the window has been changed. This is only needed for refreshes with overlapping windows.

WINDOW ***subwin(win, lines, cols, begin_y, begin_x)**
WINDOW ****win;***
int ***lines, cols, begin_y, begin_x;***

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y, begin_x*) *in the middle of the window* win. This means that any change made to either window in the area covered by the subwindow will be made on both windows. *begin_y, begin_x* are specified relative to the overall screen, not the relative (0, 0) of *win*. If either *lines* or *cols* is 0 (zero), that dimension will be set to (*LINES* − *begin_y*) or (*COLS* − *begin_x*) respectively.

unctrl(ch) †
char ***ch;***

This is actually a debug function for the library, but it is of general usefulness. It returns a string which is a representation of *ch*. Control characters become their lower-case equivalents preceded by a "^". Other letters stay just as they are. To use *unctrl()*, you must have #include <unctrl.h> in your file.

5.4. Details

gettmode()

gets the tty stats. This is normally called by *initscr()*.

mvcur(lasty, lastx, newy, newx)
int ***lasty, lastx, newy, newx;***

Moves the terminal's cursor from (*lasty, lastx*) to (*newy, newx*) in an approximation of optimal fashion. This routine uses the functions borrowed from *ex* version 2.6. It is possible to use this optimization without the benefit of the screen routines. With the screen routines, this should not be called by the user. *move()* and *refresh()* should be used to move the cursor position, so that the routines know what's going on.

savetty() †

resetty() †

Savetty() saves the current tty characteristic flags. *Resetty()* restores them to what *savetty()* stored. These functions are performed automatically by *initscr()* and *endwin()*.

setterm(name)
char ****name;***

Set the terminal characteristics to be those of the terminal named *name*. This is called by *initscr()*, so the user usually need not bother with it, unless they wish to use just the cursor motion optimizations.

1. Capabilities from termcap

1.1. Disclaimer

The description of terminals is a difficult business, and we only attempt to summarize the capabilities here: for a full description see the paper describing termcap.

1.2. Overview

Capabilities from termcap are of three kinds: string valued options, numeric valued options, and boolean options. The string valued options are the most complicated, since they may include padding information, which we describe now.

Intelligent terminals often require padding on intelligent operations at high (and sometimes even low) speed. This is specified by a number before the string in the capability, and has meaning for the capabilities which have a P at the front of their comment. This normally is a number of milliseconds to pad the operation. In the current system which has no true programmable delays, we do this by sending a sequence of pad characters (normally nulls, but can be changed (specified by *PC*)). In some cases, the pad is better computed as some number of milliseconds times the number of affected lines (to the bottom of the screen usually, except when terminals have insert modes which will shift several lines.) This is specified as, e.g., 12*. before the capability, to say 12 milliseconds per affected whatever (currently always line). Capabilities where this makes sense say P*.

1.3. Variables Set By setterm()

variables set by *setterm()*

Type	Name	Pad	Description
char *	AL	P*	Add new blank Line
bool	AM		Automatic Margins
char *	BC		Back Cursor movement
bool	BS		BackSpace works
char *	BT	P	Back Tab
bool	CA		Cursor Addressable
char *	CD	P*	Clear to end of Display
char *	CE	P	Clear to End of line
char *	CL	P*	CLear screen
char *	CM	P	Cursor Motion
char *	DC	P*	Delete Character
char *	DL	P*	Delete Line sequence
char *	DM		Delete Mode (enter)
char *	DO		DOwn line sequence
char *	ED		End Delete mode
bool	EO		can Erase Overstrikes with ' '
char *	EI		End Insert mode
char *	HO		HOme cursor
bool	HZ		HaZeltine ~ braindamage
char *	IC	P	Insert Character
bool	IN		Insert-Null blessing
char *	IM		enter Insert Mode (IC usually set, too)
char *	IP	P*	Pad after char Inserted using IM+IE

variables set by *setterm()*

Type	Name	Pad	Description
char *	LL		quick to Last Line, column 0
char *	MA		ctrl character MAp for cmd mode
bool	MI		can Move in Insert mode
bool	NC		No Cr: \r sends \r\n then eats \n
char *	ND		Non-Destructive space
bool	OS		OverStrike works
char	PC		Pad Character
char *	SE		Standout End (may leave space)
char *	SF	P	Scroll Forwards
char *	SO		Stand Out begin (may leave space)
char *	SR	P	ScRoll backwards
char *	TA	P	TAb (not ^I or with padding)
char *	TE		Terminal address enable Ending sequence
char *	TI		Terminal address enable Initialization
char *	UE		Underline Ending sequence
bool	UL		UnderLining works even though !OS
char *	UP		UPline
char *	US		Underline Starting sequence[10]
char *	VB		Visible Bell
char *	VE		Visual End sequence
char *	VS		Visual Start sequence
bool	XN		a Newline gets eaten after wrap

Names starting with *X* are reserved for severely nauseous glitches

1.4. Variables Set By gettmode()

variables set by *gettmode()*

type	name	description
bool	NONL	Term can't hack linefeeds doing a CR
bool	GT	Gtty indicates Tabs
bool	UPPERCASE	Terminal generates only uppercase letters

[10] US and UE, if they do not exist in the termcap entry, are copied from SO and SE in *initscr()*

1.
The WINDOW structure

The WINDOW structure is defined as follows:

```
# define        WINDOW  struct _win_st

struct _win_st {
        short   _cury, _curx;
        short   _maxy, _maxx;
        short   _begy, _begx;
        short   _flags;
        bool    _clear;
        bool    _leave;
        bool    _scroll;
        char    **_y;
        short   *_firstch;
        short   *_lastch;
};

# define        _SUBWIN         01
# define        _ENDLINE        02
# define        _FULLWIN        04
# define        _SCROLLWIN      010
# define        _STANDOUT       0200
```

_cury and *_curx* are the current (y, x) co-ordinates for the window. New characters added to the screen are added at this point. *_maxy* and *_maxx* are the maximum values allowed for (*_cury*, *_curx*). *_begy* and *_begx* are the starting (y, x) co-ordinates on the terminal for the window, i.e., the window's home. *_cury*, *_curx*, *_maxy*, and *_maxx* are measured relative to (*_begy*, *_begx*), not the terminal's home.

_clear tells if a clear-screen sequence is to be generated on the next *refresh()* call. This is only meaningful for screens. The initial clear-screen for the first *refresh()* call is generated by initially setting clear to be true for *curscr*, which always generates a clear-screen if set, irrelevant of the dimensions of the window involved. *_leave* is TRUE if the current (y, x) co-ordinates and the cursor are to be left after the last character changed on the terminal, or not moved if there is no change. *_scroll* is TRUE if scrolling is allowed.

_y is a pointer to an array of lines which describe the terminal. Thus:

```
_y[i]
```

is a pointer to the *i*th line, and

```
_y[i][j]
```

is the *j*th character on the *i*th line.

_flags() can have one or more values or'd into it. **_SUBWIN** means that the window is a subwindow, which indicates to *delwin()* that the space for the lines is not to be freed. **_ENDLINE** says that the end of the line for this window is also the end of a screen. **_FULLWIN** says that this window is a screen. **_SCROLLWIN** indicates that the last character of this screen is at the lower right-hand corner of the terminal; *i.e.*, if a character was put there, the terminal would scroll. **_STANDOUT** says that all characters added to the screen are in standout mode.

---

[11] All variables not normally accessed directly by the user are named with an initial "_" to avoid conflicts with the user's variables.

1. Examples

Here we present a few examples of how to use the package. They attempt to be representative, though not comprehensive.

2. Screen Updating

The following examples are intended to demonstrate the basic structure of a program using the screen updating sections of the package. Several of the programs require calculational sections which are irrelevant of to the example, and are therefore usually not included. It is hoped that the data structure definitions give enough of an idea to allow understanding of what the relevant portions do. The rest is left as an exercise to the reader, and will not be on the final.

2.1. Twinkle

This is a moderately simple program which prints pretty patterns on the screen that might even hold your interest for 30 seconds or more. It alternates between random patterns of asterisks, putting them on one by one, and then taking them off in the same fashion. It, is more efficient using only the motion optimization, as is demonstrated below.

```
# include       <curses.h>
# include       <signal.h>

/*
 * the idea for this program was a product of the imagination of
 * Kurt Schoens.  Not responsible for minds lost or stolen.
 */

# define        NCOLS   80
# define        NLINES  24
# define        MAXPATTERNS     4

struct locs {
        char    y, x;
};

typedef struct locs         LOCS;

LOCS    Layout[NCOLS * NLINES];   /* current board layout */

int     Pattern,                  /* current pattern number */
        Numstars;                 /* number of stars in pattern */

main() {

        char            *getenv();
        int             die();

        srand(getpid());                        /* initialize random sequence */

        initscr();
        signal(SIGINT, die);
        nocrmode();
        leaveok(stdscr, TRUE);
        scrollok(stdscr, FALSE);
```

```
        for (;;) {
                makeboard();            /* make the board setup */
                puton('*');             /* put on '*'s */
                puton(' ');             /* cover up with ' 's */
        }
}

/*
 * On program exit, move the cursor to the lower left corner by
 * direct addressing, since current location is not guaranteed.
 * We lie and say we used to be at the upper right corner to guarantee
 * absolute addressing.
 */
die() {

        signal(SIGINT, SIG_IGN);
        mvcur(0, COLS-1, LINES-1, 0);
        endwin();
        exit(0);
}


/*
 * Make the current board setup.  It picks a random pattern and
 * calls ison() to determine if the character is on that pattern
 * or not.
 */
makeboard() {

        reg int         y, x;
        reg LOCS        *lp;

        Pattern = rand() % MAXPATTERNS;
        lp = Layout;
        for (y = 0; y < NLINES; y++)
                for (x = 0; x < NCOLS; x++)
                        if (ison(y, x)) {
                                lp->y = y;
                                lp++->x = x;
                        }
        Numstars = lp - Layout;
}

/*
 * Return TRUE if (y, x) is on the current pattern.
 */
ison(y, x)
reg int y, x; {

        switch (Pattern) {
          case 0:               /* alternating lines */
                return !(y & 01);
```

```
                case 1:         /* box */
                        if (x >= LINES && y >= NCOLS)
                                return FALSE;
                        if (y < 3 || y >= NLINES - 3)
                                return TRUE;
                        return (x < 3 || x >= NCOLS - 3);
                case 2:         /* holy pattern! */
                        return ((x + y) & 01);
                case 3:         /* bar across center */
                        return (y >= 9 && y <= 15);
                }
                /* NOTREACHED */
}

puton(ch)
reg char        ch; {

        reg LOCS        *lp;
        reg int         r;
        reg LOCS        *end;
        LOCS            temp;

        end = &Layout[Numstars];
        for (lp = Layout; lp < end; lp++) {
                r = rand() % Numstars;
                temp = *lp;
                *lp = Layout[r];
                Layout[r] = temp;
        }

        for (lp = Layout; lp < end; lp++) {
                mvaddch(lp->y, lp->x, ch);
                refresh();
        }
}
```

2.2. Life

This program plays the famous computer pattern game of life (Scientific American, May, 1974). The calculational routines create a linked list of structures defining where each piece is. Nothing here claims to be optimal, merely demonstrative. This program, however, is a very good place to use the screen updating routines[12], as it allows them to worry about what the last position looked like, so you don't have to. It also demonstrates some of the input routines.

```
# include       <curses.h>
# include       <signal.h>

/*
 *      Run a life game.  This is a demonstration program for
 * the Screen Updating section of the -lcurses cursor package.
 */
```

[12] I bet you were beginning to wonder if there were any.

```
struct lst_st {                         /* linked list element */
        int             y, x;           /* (y, x) position of piece */
        struct lst_st   *next, *last;   /* doubly linked */
};

typedef struct lst_st   LIST;

LIST    *Head;                          /* head of linked list */

main(ac, av)
int     ac;
char    *av[]; {

        int     die();

        evalargs(ac, av);               /* evaluate arguments */

        initscr();                      /* initialize screen package */
        signal(SIGINT, die);            /* set to restore tty stats */
        crmode();                       /* set for char-by-char */
        noecho();                       /*       input */
        nonl();                         /* for optimization */

        getstart();                     /* get starting position */
        for (;;) {
                prboard();              /* print out current board */
                update();               /* update board position */
        }
}

/*
 * This is the routine which is called when rubout is hit.
 * It resets the tty stats to their original values.  This
 * is the normal way of leaving the program.
 */
die() {

        signal(SIGINT, SIG_IGN);        /* ignore rubouts */
        mvcur(0, COLS-1, LINES-1, 0);   /* go to bottom of screen */
        endwin();                       /* set terminal to initial state */
        exit(0);
}

/*
 * Get the starting position from the user.  They keys u, i, o, j, l,
 * m, ,, and . are used for moving their relative directions from the
 * k key.  Thus, u move diagonally up to the left, , moves directly down,
 * etc.  x places a piece at the current position, " " takes it away.
 * The input can also be from a file.  The list is built after the
 * board setup is ready.
 */
getstart() {

        reg char        c;
        reg int   x, y;
```

```
        box(stdscr, '|', '_');                  /* box in the screen */
        move(1, 1);                             /* move to upper left corner */

        do {
                refresh();                      /* print current position */
                if ((c=getch()) == 'q')
                        break;
                switch (c) {
                 case 'u':
                 case 'i':
                 case 'o':
                 case 'j':
                 case 'l':
                 case 'm':
                 case ',':
                 case '.':
                        adjustyx(c);
                        break;
                 case 'f':
                        mvaddstr(0, 0, "File name: ");
                        getstr(buf);
                        readfile(buf);
                        break;
                 case 'x':
                        addch('X');
                        break;
                 case ' ':
                        addch(' ');
                        break;
                }
        }

        if (Head != NULL)                       /* start new list */
                dellist(Head);
        Head = malloc(sizeof (LIST));

        /*
         * loop through the screen looking for 'x's, and add a list
         * element for each one
         */
        for (y = 1; y < LINES - 1; y++)
                for (x = 1; x < COLS - 1; x++) {
                        move(y, x);
                        if (inch() == 'x')
                                addlist(y, x);
                }
}

/*
 * Print out the current board position from the linked list
 */
prboard() {

        reg LIST        *hp;
```

```
        erase();                                /* clear out last position */
        box(stdscr, '|', '_');                  /* box in the screen */

        /*
         * go through the list adding each piece to the newly
         * blank board
         */
        for (hp = Head; hp; hp = hp->next)
                mvaddch(hp->y, hp->x, 'X');

        refresh();
}
```

3. Motion optimization

The following example shows how motion optimization is written on its own. Programs which flit from one place to another without regard for what is already there usually do not need the overhead of both space and time associated with screen updating. They should instead use motion optimization.

3.1. Twinkle

The **twinkle** program is a good candidate for simple motion optimization. Here is how it could be written:

```
main() {

        reg char        *sp;
        char            *getenv();
        int             fputchar(), die();

        srand(getpid());                        /* initialize random sequence */

        if (isatty(0)) {
            gettmode();
            if (sp=getenv("TERM"))
                    setterm(sp);
                signal(SIGINT, die);
        }
        else {
                printf("Need a terminal on %d\n", _tty_ch);
                exit(1);
        }

        noecho();
        tputs(CL, NLINES, fputchar);
        for (;;) {
                makeboard();                    /* make the board setup */
                puton('*');                     /* put on '*'s */
                puton(' ');                     /* cover up with ' 's */
        }
}

/*
 * fputchar defined for tputs()
 */
fputchar(c)
```

```
reg char        c; {

        putchar(c);
}

die() {

        signal(SIGINT, SIG_IGN);
        mvcur(0, COLS-1, LINES-1, 0);
        endwin();
        exit(0);
}


puton(ch)
char    ch; {

        static int      lasty, lastx;
        reg LOCS        *lp;
        reg int         r;
        reg LOCS        *end;
        LOCS            temp;

        end = &Layout[Numstars];
        for (lp = Layout; lp < end; lp++) {
                r = rand() % Numstars;
                temp = *lp;
                *lp = Layout[r];
                Layout[r] = temp;
        }

        for (lp = Layout; lp < end; lp++)
                /* prevent scrolling */
                if (!AM || lp->y < NLINES - 1 || lp->x < NCOLS - 1) {
                        mvcur(lasty, lastx, lp->y, lp->x);
                        putchar(ch);
                        lasty = lp->y;
                        if ((lastx = lp->x + 1) >= NCOLS)
                                if (AM) {
                                        lastx = 0;
                                        lasty++;
                                }
                                else
                                        lastx = NCOLS - 1;
                }
}
```