UNIX™

for the

68000

VOLUME III

Tutorials and Document Preparation

8/23/82

UniSoft
HOLDINGS
2405 Fourth Street • Berkeley CA 94710

VOLUME III

Tutorials and Document Preparation

Table of Contents

7th Edition UNIX — Summary

September 6, 1978

Bell Laboratories
Murray Hill, New Jersey 07974

A. What's new: highlights of the 7th edition UNIX† System

Aimed at larger systems. Devices are addressable to 2^{31} bytes, files to 2^{30} bytes. 128K memory (separate instruction and data space) is needed for some utilities.

Portability. Code of the operating system and most utilities has been extensively revised to minimize its dependence on particular hardware.

Fortran 77. F77 compiler for the new standard language is compatible with C at the object level. A Fortran structurer, STRUCT, converts old, ugly Fortran into RATFOR, a structured dialect usable with F77.

Shell. Completely new SH program supports string variables, trap handling, structured programming, user profiles, settable search path, multilevel file name generation, etc.

Document preparation. TROFF phototypesetter utility is standard. NROFF (for terminals) is now highly compatible with TROFF. MS macro package provides canned commands for many common formatting and layout situations. TBL provides an easy to learn language for preparing complicated tabular material. REFER fills in bibliographic citations from a data base.

UNIX-to-UNIX file copy. UUCP performs spooled file transfers between any two machines.

Data processing. SED stream editor does multiple editing functions in parallel on a data stream of indefinite length. AWK report generator does free-field pattern selection and arithmetic operations.

Program development. MAKE controls re-creation of complicated software, arranging for minimal recompilation.

Debugging. ADB does postmortem and breakpoint debugging, handles separate instruction and data spaces, floating point, etc.

C language. The language now supports definable data types, generalized initialization, block structure, long integers, unions, explicit type conversions. The LINT verifier does strong type checking and detection of probable errors and portability problems even across separately compiled functions.

Lexical analyzer generator. LEX converts specification of regular expressions and semantic actions into a recognizing subroutine. Analogous to YACC.

Graphics. Simple graph-drawing utility, graphic subroutines, and generalized plotting filters adapted to various devices are now standard.

Standard input-output package. Highly efficient buffered stream I/O is integrated with formatted input and output.

Other. The operating system and utilities have been enhanced and freed of restrictions in many other ways too numerous to relate.

---

† UNIX is a Trademark of Bell Laboratories.

B. Hardware

The 7th edition UNIX operating system runs on a DEC PDP-11/45 or 11/70* with at least the following equipment:

128K to 2M words of managed memory; parity not used.

disk: RP03, RP04, RP06, RK05 (more than 1 RK05) or equivalent.

console typewriter.

clock: KW11-L or KW11-P.

The following equipment is strongly recommended:

communications controller such as DL11 or DH11.

full duplex 96-character ASCII terminals.

9-track tape or extra disk for system backup.

The system is normally distributed on 9-track tape. The minimum memory and disk space specified is enough to run and maintain UNIX. More will be needed to keep all source on line, or to handle a large number of users, big data bases, diversified complements of devices, or large programs. The resident code occupies 12-20K words depending on configuration; system data occupies 10-28K words.

There is no commitment to provide 7th edition UNIX on PDP-11/34, 11/40 and 11/60 hardware.

C. Software

Most of the programs available as UNIX commands are listed. Source code and printed manuals are distributed for all of the listed software except games. Almost all of the code is written in C. Commands are self-contained and do not require extra setup information, unless specifically noted as "interactive." Interactive programs can be made to run from a prepared script simply by redirecting input. Most programs intended for interactive use (e.g., the editor) allow for an escape to command level (the Shell). Most file processing commands can also go from standard input to standard output ("filters"). The piping facility of the Shell may be used to connect such filters directly to the input or output of other programs.

1. Basic Software

This includes the time-sharing operating system with utilities, a machine language assembler and a compiler for the programming language C—enough software to write and run new applications and to maintain or modify UNIX itself.

1.1. Operating System

□ UNIX — The basic resident code on which everything else depends. Supports the system calls, and maintains the file system. A general description of UNIX design philosophy and system facilities appeared in the Communications of the ACM, July, 1974. A more extensive survey is in the Bell System Technical Journal for July-August 1978. Capabilities include:

- ○ Reentrant code for user processes.
- ○ Separate instruction and data spaces.
- ○ "Group" access permissions for cooperative projects, with overlapping memberships.
- ○ Alarm-clock timeouts.

*PDP is a Trademark of Digital Equipment Corporation.

- Timer-interrupt sampling and interprocess monitoring for debugging and measurement.
- Multiplexed I/O for machine-to-machine communication.

□ DEVICES All I/O is logically synchronous. I/O devices are simply files in the file system. Normally, invisible buffering makes all physical record structure and device characteristics transparent and exploits the hardware's ability to do overlapped I/O. Unbuffered physical record I/O is available for unusual applications. Drivers for these devices are available; others can be easily written:

- Asynchronous interfaces: DH11, DL11. Support for most common ASCII terminals.
- Synchronous interface: DP11.
- Automatic calling unit interface: DN11.
- Line printer: LP11.
- Magnetic tape: TU10 and TU16.
- DECtape: TC11.
- Fixed head disk: RS11, RS03 and RS04.
- Pack type disk: RP03, RP04, RP06; minimum-latency seek scheduling.
- Cartridge-type disk: RK05, one or more physical devices per logical device.
- Null device.
- Physical memory of PDP-11, or mapped memory in resident system.
- Phototypesetter: Graphic Systems System/1 through DR11C.

□ BOOT Procedures to get UNIX started.

□ MKCONF Tailor device-dependent system code to hardware configuration. As distributed, UNIX can be brought up directly on any acceptable CPU with any acceptable disk, any sufficient amount of core, and either clock. Other changes, such as optimal assignment of directories to devices, inclusion of floating point simulator, or installation of device names in file system, can then be made at leisure.

1.2. User Access Control

□ LOGIN Sign on as a new user.

- Verify password and establish user's individual and group (project) identity.
- Adapt to characteristics of terminal.
- Establish working directory.
- Announce presence of mail (from MAIL).
- Publish message of the day.
- Execute user-specified profile.
- Start command interpreter or other initial program.

□ PASSWD Change a password.

- User can change his own password.
- Passwords are kept encrypted for security.

□ NEWGRP Change working group (project). Protects against unauthorized changes to projects.

1.3. Terminal Handling

□ TABS Set tab stops appropriately for specified terminal type.

□ STTY Set up options for optimal control of a terminal. In so far as they are deducible from the input, these options are set automatically by LOGIN.

- O Half vs. full duplex.
- O Carriage return+line feed vs. newline.
- O Interpretation of tabs.
- O Parity.
- O Mapping of upper case to lower.
- O Raw vs. edited input.
- O Delays for tabs, newlines and carriage returns.

1.4. File Manipulation

- ☐ CAT — Concatenate one or more files onto standard output. Particularly used for unadorned printing, for inserting data into a pipeline, and for buffering output that comes in dribs and drabs. Works on any file regardless of contents.
- ☐ CP — Copy one file to another, or a set of files to a directory. Works on any file regardless of contents.
- ☐ PR — Print files with title, date, and page number on every page.
  - O Multicolumn output.
  - O Parallel column merge of several files.
- ☐ LPR — Off-line print. Spools arbitrary files to the line printer.
- ☐ CMP — Compare two files and report if different.
- ☐ TAIL — Print last *n* lines of input
  - O May print last *n* characters, or from *n* lines or characters to end.
- ☐ SPLIT — Split a large file into more manageable pieces. Occasionally necessary for editing (ED).
- ☐ DD — Physical file format translator, for exchanging data with foreign systems, especially IBM 370's.
- ☐ SUM — Sum the words of a file.

1.5. Manipulation of Directories and File Names

- ☐ RM — Remove a file. Only the name goes away if any other names are linked to the file.
  - O Step through a directory deleting files interactively.
  - O Delete entire directory hierarchies.
- ☐ LN — "Link" another name (alias) to an existing file.
- ☐ MV — Move a file or files. Used for renaming files.
- ☐ CHMOD — Change permissions on one or more files. Executable by files' owner.
- ☐ CHOWN — Change owner of one or more files.
- ☐ CHGRP — Change group (project) to which a file belongs.
- ☐ MKDIR — Make a new directory.
- ☐ RMDIR — Remove a directory.
- ☐ CD — Change working directory.
- ☐ FIND — Prowl the directory hierarchy finding every file that meets specified criteria.

- O Criteria include:
  - name matches a given pattern,
  - creation date in given range,
  - date of last use in given range,
  - given permissions,
  - given owner,
  - given special file characteristics,
  - boolean combinations of above.
- O Any directory may be considered to be the root.
- O Perform specified command on each file found.

1.6. Running of Programs

□ SH — The Shell, or command language interpreter.
- O Supply arguments to and run any executable program.
- O Redirect standard input, standard output, and standard error files.
- O Pipes: simultaneous execution with output of one process connected to the input of another.
- O Compose compound commands using:
  - if ... then ... else,
  - case switches,
  - while loops,
  - for loops over lists,
  - break, continue and exit,
  - parentheses for grouping.
- O Initiate background processes.
- O Perform Shell programs, i.e., command scripts with substitutable arguments.
- O Construct argument lists from all file names satisfying specified patterns.
- O Take special action on traps and interrupts.
- O User-settable search path for finding commands.
- O Executes user-settable profile upon login.
- O Optionally announces presence of mail as it arrives.
- O Provides variables and parameters with default setting.

□ TEST — Tests for use in Shell conditionals.
- O String comparison.
- O File nature and accessibility.
- O Boolean combinations of the above.

□ EXPR — String computations for calculating command arguments.
- O Integer arithmetic
- O Pattern matching

□ WAIT — Wait for termination of asynchronously running processes.

□ READ — Read a line from terminal, for interactive Shell procedure.

□ ECHO — Print remainder of command line. Useful for diagnostics or prompts in Shell programs, or for inserting data into a pipeline.

□ SLEEP — Suspend execution for a specified time.

□ NOHUP — Run a command immune to hanging up the terminal.

□ NICE — Run a command in low (or high) priority.

- KILL — Terminate named processes.
- CRON — Schedule regular actions at specified times.
  - Actions are arbitrary programs.
  - Times are conjunctions of month, day of month, day of week, hour and minute. Ranges are specifiable for each.
- AT — Schedule a one-shot action for an arbitrary time.
- TEE — Pass data between processes and divert a copy into one or more files.

1.7. Status Inquiries

- LS — List the names of one, several, or all files in one or more directories.
  - Alphabetic or temporal sorting, up or down.
  - Optional information: size, owner, group, date last modified, date last accessed, permissions, i-node number.
- FILE — Try to determine what kind of information is in a file by consulting the file system index and by reading the file itself.
- DATE — Print today's date and time. Has considerable knowledge of calendric and horological peculiarities.
  - May set UNIX's idea of date and time.
- DF — Report amount of free space on file system devices.
- DU — Print a summary of total space occupied by all files in a hierarchy.
- QUOT — Print summary of file space usage by user id.
- WHO — Tell who's on the system.
  - List of presently logged in users, ports and times on.
  - Optional history of all logins and logouts.
- PS — Report on active processes.
  - List your own or everybody's processes.
  - Tell what commands are being executed.
  - Optional status information: state and scheduling info, priority, attached terminal, what it's waiting for, size.
- IOSTAT — Print statistics about system I/O activity.
- TTY — Print name of your terminal.
- PWD — Print name of your working directory.

1.8. Backup and Maintenance

- MOUNT — Attach a device containing a file system to the tree of directories. Protects against nonsense arrangements.
- UMOUNT — Remove the file system contained on a device from the tree of directories. Protects against removing a busy device.
- MKFS — Make a new file system on a device.
- MKNOD — Make an i-node (file system entry) for a special file. Special files are physical devices, virtual devices, physical memory, etc.

□ TP

□ TAR — Manage file archives on magnetic tape or DECtape. TAR is newer.
- ○ Collect files into an archive.
- ○ Update DECtape archive by date.
- ○ Replace or delete DECtape files.
- ○ Print table of contents.
- ○ Retrieve from archive.

□ DUMP — Dump the file system stored on a specified device, selectively by date, or indiscriminately.

□ RESTOR — Restore a dumped file system, or selectively retrieve parts thereof.

□ SU — Temporarily become the super user with all the rights and privileges thereof. Requires a password.

□ DCHECK

□ ICHECK

□ NCHECK — Check consistency of file system.
- ○ Print gross statistics: number of files, number of directories, number of special files, space used, space free.
- ○ Report duplicate use of space.
- ○ Retrieve lost space.
- ○ Report inaccessible files.
- ○ Check consistency of directories.
- ○ List names of all files.

□ CLRI — Peremptorily expunge a file and its space from a file system. Used to repair damaged file systems.

□ SYNC — Force all outstanding I/O on the system to completion. Used to shut down gracefully.

1.9. Accounting

The timing information on which the reports are based can be manually cleared or shut off completely.

□ AC — Publish cumulative connect time report.
- ○ Connect time by user or by day.
- ○ For all users or for selected users.

□ SA — Publish Shell accounting report. Gives usage information on each command executed.
- ○ Number of times used.
- ○ Total system time, user time and elapsed time.
- ○ Optional averages and percentages.
- ○ Sorting on various fields.

1.10. Communication

□ MAIL — Mail a message to one or more users. Also used to read and dispose of incoming mail. The presence of mail is announced by LOGIN and optionally by SH.
- ○ Each message can be disposed of individually.
- ○ Messages can be saved in files or forwarded.

- ☐ CALENDAR Automatic reminder service for events of today and tomorrow.
- ☐ WRITE Establish direct terminal communication with another user.
- ☐ WALL Write to all users.
- ☐ MESG Inhibit receipt of messages from WRITE and WALL.
- ☐ CU Call up another time-sharing system.
  - ○ Transparent interface to remote machine.
  - ○ File transmission.
  - ○ Take remote input from local file or put remote output into local file.
  - ○ Remote system need not be UNIX.
- ☐ UUCP UNIX to UNIX copy.
  - ○ Automatic queuing until line becomes available and remote machine is up.
  - ○ Copy between two remote machines.
  - ○ Differences, mail, etc., between two machines.

1.11. Basic Program Development Tools

Some of these utilities are used as integral parts of the higher level languages described in section 2.

- ☐ AR Maintain archives and libraries. Combines several files into one for housekeeping efficiency.
  - ○ Create new archive.
  - ○ Update archive by date.
  - ○ Replace or delete files.
  - ○ Print table of contents.
  - ○ Retrieve from archive.
- ☐ AS Assembler. Similar to PAL-11, but different in detail.
  - ○ Creates object program consisting of
    code, possibly read-only,
    initialized data or read-write code,
    uninitialized data.
  - ○ Relocatable object code is directly executable without further transformation.
  - ○ Object code normally includes a symbol table.
  - ○ Multiple source files.
  - ○ Local labels.
  - ○ Conditional assembly.
  - ○ "Conditional jump" instructions become branches or branches plus jumps depending on distance.
- ☐ Library The basic run-time library. These routines are used freely by all software.
  - ○ Buffered character-by-character I/O.
  - ○ Formatted input and output conversion (SCANF and PRINTF) for standard input and output, files, in-memory conversion.
  - ○ Storage allocator.
  - ○ Time conversions.
  - ○ Number conversions.
  - ○ Password encryption.
  - ○ Quicksort.
  - ○ Random number generator.
  - ○ Mathematical function library, including trigonometric functions and inverses, exponential, logarithm, square root, bessel functions.

⊏ ADB — Interactive debugger.
- C Postmortem dumping.
- O Examination of arbitrary files, with no limit on size.
- O Interactive breakpoint debugging with the debugger as a separate process.
- O Symbolic reference to local and global variables.
- O Stack trace for C programs.
- O Output formats:
  - 1-, 2-, or 4-byte integers in octal, decimal, or hex
  - single and double floating point
  - character and string
  - disassembled machine instructions
- O Patching.
- O Searching for integer, character, or floating patterns.
- O Handles separated instruction and data space.

⊏ OD — Dump any file. Output options include any combination of octal or decimal by words, octal by bytes, ASCII, opcodes, hexadecimal.
- O Range of dumping is controllable.

⊏ LD — Link edit. Combine relocatable object files. Insert required routines from specified libraries.
- O Resulting code may be sharable.
- O Resulting code may have separate instruction and data spaces.

⊏ LORDER — Places object file names in proper order for loading, so that files depending on others come after them.

⊏ NM — Print the namelist (symbol table) of an object program. Provides control over the style and order of names that are printed.

⊏ SIZE — Report the core requirements of one or more object files.

⊏ STRIP — Remove the relocation and symbol table information from an object file to save space.

⊏ TIME — Run a command and report timing information on it.

⊏ PROF — Construct a profile of time spent per routine from statistics gathered by time-sampling the execution of a program. Uses floating point.
- O Subroutine call frequency and average times for C programs.

⊏ MAKE — Controls creation of large programs. Uses a control file specifying source file dependencies to make new version; uses time last changed to deduce minimum amount of work necessary.
- O Knows about CC, YACC, LEX, etc.

1.12. UNIX Programmer's Manual

⊏ Manual — Machine-readable version of the UNIX Programmer's Manual.
- O System overview.
- O All commands.
- O All system calls.
- O All subroutines in C and assembler libraries.
- O All devices and other special files.
- O Formats of file system and kinds of files known to system software.
- ⊃ Boot and maintenance procedures.

□ MAN Print specified manual section on your terminal.

1.13. Computer-Aided Instruction

□ LEARN A program for interpreting CAI scripts, plus scripts for learning about UNIX by using it.
- O Scripts for basic files and commands, editor, advanced files and commands, EQN, MS macros, C programming language.

2. Languages

2.1. The C Language

□ CC Compile and/or link edit programs in the C language. The UNIX operating system, most of the subsystems and C itself are written in C. For a full description of C, read *The C Programming Language*, Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, 1978.
- O General purpose language designed for structured programming.
- O Data types include character, integer, float, double, pointers to all types, functions returning above types, arrays of all types, structures and unions of all types.
- O Operations intended to give machine-independent control of full machine facility, including to-memory operations and pointer arithmetic.
- O Macro preprocessor for parameterized code and inclusion of standard files.
- O All procedures recursive, with parameters by value.
- O Machine-independent pointer manipulation.
- O Object code uses full addressing capability of the PDP-11.
- O Runtime library gives access to all system facilities.
- O Definable data types.
- O Block structure

□ LINT Verifier for C programs. Reports questionable or nonportable usage such as:
  Mismatched data declarations and procedure interfaces.
  Nonportable type conversions.
  Unused variables, unreachable code, no-effect operations.
  Mistyped pointers.
  Obsolete syntax.
- O Full cross-module checking of separately compiled programs.

□ CB A beautifier for C programs. Does proper indentation and placement of braces.

2.2. Fortran

□ F77 A full compiler for ANSI Standard Fortran 77.
- O Compatible with C and supporting tools at object level.
- O Optional source compatibility with Fortran 66.
- O Free format source.
- O Optional subscript-range checking, detection of uninitialized variables.
- O All widths of arithmetic: 2- and 4-byte integer; 4- and 8-byte real; 8- and 16-byte complex.

□ RATFOR Ratfor adds rational control structure à la C to Fortran.
- O Compound statements.

- O If-else, do, for, while, repeat-until, break, next statements.
- O Symbolic constants.
- O File insertion.
- O Free format source
- O Translation of relationals like >, >=.
- O Produces genuine Fortran to carry away.
- O May be used with F77.

□ STRUCT — Converts ordinary ugly Fortran into structured Fortran (i.e., Ratfor), using statement grouping, if-else, while, for, repeat-until.

2.3. Other Algorithmic Languages

□ BAS — An interactive interpreter, similar in style to BASIC. Interpret unnumbered statements immediately, numbered statements upon 'run'.

- O Statements include:
  - comment,
  - dump,
  - for...next,
  - goto,
  - if...else...fi,
  - list,
  - print,
  - prompt,
  - return,
  - run,
  - save.
- O All calculations double precision.
- O Recursive function defining and calling.
- O Builtin functions include log, exp, sin, cos, atn, int, sqr, abs, rnd.
- O Escape to ED for complex program editing.

□ DC — Interactive programmable desk calculator. Has named storage locations as well as conventional stack for holding integers or programs.

- O Unlimited precision decimal arithmetic.
- O Appropriate treatment of decimal fractions.
- O Arbitrary input and output radices, in particular binary, octal, decimal and hexadecimal.
- O Reverse Polish operators:
  - + − * /
  - remainder, power, square root,
  - load, store, duplicate, clear,
  - print, enter program text, execute.

□ BC — A C-like interactive interface to the desk calculator DC.

- O All the capabilities of DC with a high-level syntax.
- O Arrays and recursive functions.
- O Immediate evaluation of expressions and evaluation of functions upon call.
- O Arbitrary precision elementary functions: exp, sin, cos, atan.
- O Go-to-less programming.

2.4. Macroprocessing

□ M4 — A general purpose macroprocessor.
- O Stream-oriented, recognizes macros anywhere in text.
- O Syntax fits with functional syntax of most higher-level languages.
- O Can evaluate integer arithmetic expressions.

2.5. Compiler-compilers

□ YACC — An LR(1)-based compiler writing system. During execution of resulting parsers, arbitrary C functions may be called to do code generation or semantic actions.
- O BNF syntax specifications.
- O Precedence relations.
- O Accepts formally ambiguous grammars with non-BNF resolution rules.

□ LEX — Generator of lexical analyzers. Arbitrary C functions may be called upon isolation of each lexical token.
- O Full regular expression, plus left and right context dependence.
- O Resulting lexical analysers interface cleanly with YACC parsers.

3. Text Processing

3.1. Document Preparation

□ ED — Interactive context editor. Random access to all lines of a file.
- O Find lines by number or pattern. Patterns may include: specified characters, don't care characters, choices among characters, repetitions of these constructs, beginning of line, end of line.
- O Add, delete, change, copy, move or join lines.
- O Permute or split contents of a line.
- O Replace one or all instances of a pattern within a line.
- O Combine or split files.
- O Escape to Shell (command language) during editing.
- O Do any of above operations on every pattern-selected line in a given range.
- O Optional encryption for extra security.

□ PTX — Make a permuted (key word in context) index.

□ SPELL — Look for spelling errors by comparing each word in a document against a word list.
- O 25,000-word list includes proper names.
- O Handles common prefixes and suffixes.
- O Collects words to help tailor local spelling lists.

□ LOOK — Search for words in dictionary that begin with specified prefix.

□ TYPO — Look for spelling errors by a statistical technique; not limited to English.

□ CRYPT — Encrypt and decrypt files for security.

3.2. Document Formatting

□ ROFF — A typesetting program for terminals. Easy for nontechnical people to learn, and good for simple documents. Input consists of data lines intermixed with control lines, such as

```
.sp 2     insert two lines of space
.ce       center the next line
```

ROFF is deemed to be obsolete; it is intended only for casual use.

- O Justification of either or both margins.
- O Automatic hyphenation.
- O Generalized running heads and feet, with even-odd page capability, numbering, etc.
- O Definable macros for frequently used control sequences (no substitutable arguments).
- O All 4 margins and page size dynamically adjustable.
- O Hanging indents and one-line indents.
- O Absolute and relative parameter settings.
- O Optional legal-style numbering of output lines.
- O Multiple file capability.
- O Not usable as a filter.

□ TROFF

□ NROFF Advanced typesetting. TROFF drives a Graphic Systems phototypesetter; NROFF drives ascii terminals of all types. This summary was typeset using TROFF. TROFF and NROFF style is similar to ROFF, but they are capable of much more elaborate feats of formatting, when appropriately programmed. TROFF and NROFF accept the same input language.

- O All ROFF capabilities available or definable.
- O Completely definable page format keyed to dynamically planted "interrupts" at specified lines.
- O Maintains several separately definable typesetting environments (e.g., one for body text, one for footnotes, and one for unusually elaborate headings).
- O Arbitrary number of output pools can be combined at will.
- O Macros with substitutable arguments, and macros invocable in mid-line.
- O Computation and printing of numerical quantities.
- O Conditional execution of macros.
- O Tabular layout facility.
- O Positions expressible in inches, centimeters, ems, points, machine units or arithmetic combinations thereof.
- O Access to character-width computation for unusually difficult layout problems.
- O Overstrikes, built-up brackets, horizontal and vertical line drawing.
- O Dynamic relative or absolute positioning and size selection, globally or at the character level.
- O Can exploit the characteristics of the terminal being used, for approximating special characters, reverse motions, proportional spacing, etc.

The Graphic Systems typesetter has a vocabulary of several 102-character fonts (4 simultaneously) in 15 sizes. TROFF provides terminal output for rough sampling of the product.

NROFF will produce multicolumn output on terminals capable of reverse line feed, or through the postprocessor COL.

High programming skill is required to exploit the formatting capabilities of TROFF and NROFF, although unskilled personnel can easily be trained to enter documents according to canned formats such as those provided by MS, below. TROFF and EQN are essentially identical to NROFF and NEQN so it is usually possible to define interchangeable formats to produce approximate proof copy on terminals before actual typesetting. The preprocessors MS, TBL, and REFER are fully compatible with TROFF and NROFF.

□ MS A standardized manuscript layout package for use with NROFF/TROFF. This document was formatted with MS.

- O Page numbers and draft dates.
- O Automatically numbered subheads.
- O Footnotes.
- O Single or double column.
- O Paragraphing, display and indentation.
- O Numbered equations.

□ EQN — A mathematical typesetting preprocessor for TROFF. Translates easily readable formulas, either in-line or displayed, into detailed typesetting instructions. Formulas are written in a style like this:

sigma sup 2 "=" 1 over N sum from i=1 to N (x sub i − x bar) sup 2

which produces:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i-\bar{x})^2$$

- O Automatic calculation of size changes for subscripts, sub-subscripts, etc.
- O Full vocabulary of Greek letters and special symbols, such as 'gamma', 'GAMMA', 'integral'.
- O Automatic calculation of large bracket sizes.
- O Vertical "piling" of formulae for matrices, conditional alternatives, etc.
- O Integrals, sums, etc., with arbitrarily complex limits.
- O Diacriticals: dots, double dots, hats, bars, etc.
- O Easily learned by nonprogrammers and mathematical typists.

□ NEQN — A version of EQN for NROFF; accepts the same input language. Prepares formulas for display on any terminal that NROFF knows about, for example, those based on Diablo printing mechanism.
- O Same facilities as EQN within graphical capability of terminal.

□ TBL — A preprocessor for NROFF/TROFF that translates simple descriptions of table layouts and contents into detailed typesetting instructions.
- O Computes column widths.
- O Handles left- and right-justified columns, centered columns and decimal-point alignment.
- O Places column titles.
- O Table entries can be text, which is adjusted to fit.
- O Can box all or parts of table.

□ REFER — Fills in bibliographic citations in a document from a data base (not supplied).
- O References may be printed in any style, as they occur or collected at the end.
- O May be numbered sequentially, by name of author, etc.

□ TC — Simulate Graphic Systems typesetter on Tektronix 4014 scope. Useful for checking TROFF page layout before typesetting.

□ GREEK — Fancy printing on Diablo-mechanism terminals like DASI-300 and DASI-450, and on Tektronix 4014.
- O Gives half-line forward and reverse motions.
- O Approximates Greek letters and other special characters by overstriking.

□ COL — Canonicalize files with reverse line feeds for one-pass printing.

□ DEROFF — Remove all TROFF commands from input.

□ CHECKEQ — Check document for possible errors in EQN usage.

4. Information Handling

- □ SORT — Sort or merge ASCII files line-by-line. No limit on input size.
  - O Sort up or down.
  - O Sort lexicographically or on numeric key.
  - O Multiple keys located by delimiters or by character position.
  - O May sort upper case together with lower into dictionary order.
  - O Optionally suppress duplicate data.
- □ TSORT — Topological sort — converts a partial order into a total order.
- □ UNIQ — Collapse successive duplicate lines in a file into one line.
  - O Publish lines that were originally unique, duplicated, or both.
  - O May give redundancy count for each line.
- □ TR — Do one-to-one character translation according to an arbitrary code.
  - O May coalesce selected repeated characters.
  - O May delete selected characters.
- □ DIFF — Report line changes, additions and deletions necessary to bring two files into agreement.
  - O May produce an editor script to convert one file into another.
  - O A variant compares two new versions against one old one.
- □ COMM — Identify common lines in two sorted files. Output in up to 3 columns shows lines present in first file only, present in both, and/or present in second only.
- □ JOIN — Combine two files by joining records that have identical keys.
- □ GREP — Print all lines in a file that satisfy a pattern as used in the editor ED.
  - O May print all lines that fail to match.
  - O May print count of hits.
  - O May print first hit in each file.
- □ LOOK — Binary search in sorted file for lines with specified prefix.
- □ WC — Count the lines, "words" (blank-separated strings) and characters in a file.
- □ SED — Stream-oriented version of ED. Can perform a sequence of editing operations on each line of an input stream of unbounded length.
  - O Lines may be selected by address or range of addresses.
  - O Control flow and conditional testing.
  - O Multiple output streams.
  - O Multi-line capability.
- □ AWK — Pattern scanning and processing language. Searches input for patterns, and performs actions on each line of input that satisfies the pattern.
  - O Patterns include regular expressions, arithmetic and lexicographic conditions, boolean combinations and ranges of these.
  - O Data treated as string or numeric as appropriate.
  - O Can break input into fields; fields are variables.
  - O Variables and arrays (with non-numeric subscripts).
  - O Full set of arithmetic operators and control flow.
  - O Multiple output streams to files and pipes.
  - O Output can be formatted as desired.
  - O Multi-line capabilities.

5. Graphics

The programs in this section are predominantly intended for use with Tektronix 4014 storage scopes.

- ☐ GRAPH — Prepares a graph of a set of input numbers.
  - ○ Input scaled to fit standard plotting area.
  - ○ Abscissae may be supplied automatically.
  - ○ Graph may be labeled.
  - ○ Control over grid style, line style, graph orientation, etc.
- ☐ SPLINE — Provides a smooth curve through a set of points intended for GRAPH.
- ☐ PLOT — A set of filters for printing graphs produced by GRAPH and other programs on various terminals. Filters provided for 4014, DASI terminals, Versatec printer/plotter.

6. Novelties, Games, and Things That Didn't Fit Anywhere Else

- ☐ BACKGAMMON — A player of modest accomplishment.
- ☐ CHESS — Plays good class D chess.
- ☐ CHECKERS — Ditto, for checkers.
- ☐ BCD — Converts ascii to card-image form.
- ☐ PPT — Converts ascii to paper tape form.
- ☐ BJ — A blackjack dealer.
- ☐ CUBIC — An accomplished player of 4×4×4 tic-tac-toe.
- ☐ MAZE — Constructs random mazes for you to solve.
- ☐ MOO — A fascinating number-guessing game.
- ☐ CAL — Print a calendar of specified month and year.
- ☐ BANNER — Print output in huge letters.
- ☐ CHING — The *I Ching*. Place your own interpretation on the output.
- ☐ FORTUNE — Presents a random fortune cookie on each invocation. Limited jar of cookies included.
- ☐ UNITS — Convert amounts between different scales of measurement. Knows hundreds of units. For example, how many km/sec is a parsec/megayear?
- ☐ TTT — A tic-tac-toe program that learns. It never makes the same mistake twice.
- ☐ ARITHMETIC — Speed and accuracy test for number facts.
- ☐ FACTOR — Factor large integers.
- ☐ QUIZ — Test your knowledge of Shakespeare, Presidents, capitals, etc.
- ☐ WUMP — Hunt the wumpus, thrilling search in a dangerous cave.
- ☐ REVERSI — A two person board game, isomorphic to Othello®.
- ☐ HANGMAN — Word-guessing game. Uses the dictionary supplied with SPELL.

□ FISH Children's card-guessing game.

The UNIX Time-Sharing System*

D. M. Ritchie and K. Thompson

ABSTRACT

UNIX† is a general-purpose, multi-user, interactive operating system for the larger Digital Equipment Corporation PDP-11 and the Interdata 8/32 computers. It offers a number of features seldom found even in larger operating systems, including

i A hierarchical file system incorporating demountable volumes,

ii Compatible file, device, and inter-process I/O,

iii The ability to initiate asynchronous processes,

iv System command language selectable on a per-user basis,

v Over 100 subsystems including a dozen languages,

vi High degree of portability.

This paper discusses the nature and implementation of the file system and of the user command interface.

1. INTRODUCTION

There have been four versions of the UNIX time-sharing system. The earliest (circa 1969-70) ran on the Digital Equipment Corporation PDP-7 and -9 computers. The second version ran on the unprotected PDP-11/20 computer. The third incorporated multiprogramming and ran on the PDP-11/34, /40, /45, /60, and /70 computers; it is the one described in the previously published version of this paper, and is also the most widely used today. This paper describes only the fourth, current system that runs on the PDP-11/70 and the Interdata 8/32 computers. In fact, the differences among the various systems is rather small; most of the revisions made to the originally published version of this paper, aside from those concerned with style, had to do with details of the implementation of the file system.

Since PDP-11 UNIX became operational in February, 1971, over 600 installations have been put into service. Most of them are engaged in applications such as computer science education, the preparation and formatting of documents and other textual material, the collection and processing of trouble data from various switching machines within the Bell System, and recording and checking telephone service orders. Our own installation is used mainly for research in operating systems, languages, computer networks, and other topics in computer science, and also for document preparation.

Perhaps the most important achievement of UNIX is to demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort: it can run on hardware costing as little as $40,000, and less than two man-years were spent on the main system software. We hope, however, that users find that the most important

 This is a revised version of an article that appeared in Communications of the ACM, *17*, No. 7 (July 1974), pp. 365-375. That article was a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15-17, 1973.

characteristics of the system are its simplicity, elegance, and ease of use.

Besides the operating system proper, some major programs available under UNIX are

C compiler
Text editor based on QED[1]
Assembler, linking loader, symbolic debugger
Phototypesetting and equation setting programs[2,3]
Dozens of languages including Fortran 77, Basic, Snobol, APL, Algol 68, M6, TMG, Pascal

There is a host of maintenance, utility, recreation and novelty programs, all written locally. The UNIX user community, which numbers in the thousands, has contributed many more programs and languages. It is worth noting that the system is totally self-supporting. All UNIX software is maintained on the system; likewise, this paper and all other documents in this issue were generated and formatted by the UNIX editor and text formatting programs.

II. HARDWARE AND SOFTWARE ENVIRONMENT

The PDP-11/70 on which the Research UNIX system is installed is a 16-bit word (8-bit byte) computer with 768K bytes of core memory; the system kernel occupies 90K bytes about equally divided between code and data tables. This system, however, includes a very large number of device drivers and enjoys a generous allotment of space for I/O buffers and system tables; a minimal system capable of running the software mentioned above can require as little as 96K bytes of core altogether. There are even larger installations; see the description of the PWB/UNIX systems,[4,5] for example. There are also much smaller, though somewhat restricted, versions of the system.[6]

Our own PDP-11 has two 200-Mb moving-head disks for file system storage and swapping. There are 20 variable-speed communications interfaces attached to 300- and 1200-baud data sets, and an additional 12 communication lines hard-wired to 9600-baud terminals and satellite computers. There are also several 2400- and 4800-baud synchronous communication interfaces used for machine-to-machine file transfer. Finally, there is a variety of miscellaneous devices including nine-track magnetic tape, a line printer, a voice synthesizer, a phototypesetter, a digital switching network, and a chess machine.

The preponderance of UNIX software is written in the abovementioned C language.[7] Early versions of the operating system were written in assembly language, but during the summer of 1973, it was rewritten in C. The size of the new system was about one-third greater than that of the old. Since the new system not only became much easier to understand and to modify but also included many functional improvements, including multiprogramming and the ability to share reentrant code among several user programs, we consider this increase in size quite acceptable.

III. THE FILE SYSTEM

The most important role of the system is to provide a file system. From the point of view of the user, there are three kinds of files: ordinary disk files, directories, and special files.

3.1 Ordinary files

A file contains whatever information the user places on it, for example, symbolic or binary (object) programs. No particular structuring is expected by the system. A file of text consists simply of a string of characters, with lines demarcated by the newline character. Binary programs are sequences of words as they will appear in core memory when the program starts executing. A few user programs manipulate files with more structure; for example, the assembler generates, and the loader expects, an object file in a particular format. However, the structure of files is controlled by the programs that use them, not by the system.

3.2 Directories

Directories provide the mapping between the names of files and the files themselves, and thus induce a structure on the file system as a whole. Each user has a directory of his own files; he may also create subdirectories to contain groups of files conveniently treated together. A directory behaves exactly like an ordinary file except that it cannot be written on by unprivileged programs, so that the system controls the contents of directories. However, anyone with appropriate permission may read a directory just like any other file.

The system maintains several directories for its own use. One of these is the **root** directory. All files in the system can be found by tracing a path through a chain of directories until the desired file is reached. The starting point for such searches is often the **root**. Other system directories contain all the programs provided for general use; that is, all the *commands*. As will be seen, however, it is by no means necessary that a program reside in one of these directories for it to be executed.

Files are named by sequences of 14 or fewer characters. When the name of a file is specified to the system, it may be in the form of a *path name*, which is a sequence of directory names separated by slashes, "/", and ending in a file name. If the sequence begins with a slash, the search begins in the root directory. The name **/alpha/beta/gamma** causes the system to search the root for directory **alpha**, then to search **alpha** for **beta**, finally to find **gamma** in **beta**. **gamma** may be an ordinary file, a directory, or a special file. As a limiting case, the name "/" refers to the root itself.

A path name not starting with "/" causes the system to begin the search in the user's current directory. Thus, the name **alpha/beta** specifies the file named **beta** in subdirectory **alpha** of the current directory. The simplest kind of name, for example, **alpha**, refers to a file that itself is found in the current directory. As another limiting case, the null file name refers to the current directory.

The same non-directory file may appear in several directories under possibly different names. This feature is called *linking*, a directory entry for a file is sometimes called a link. The UNIX system differs from other systems in which linking is permitted in that all links to a file have equal status. That is, a file does not exist within a particular directory; the directory entry for a file consists merely of its name and a pointer to the information actually describing the file. Thus a file exists independently of any directory entry, although in practice a file is made to disappear along with the last link to it.

Each directory always has at least two entries. The name "." in each directory refers to the directory itself. Thus a program may read the current directory under the name "." without knowing its complete path name. The name ".." by convention refers to the parent of the directory in which it appears, that is, to the directory in which it was created.

The directory structure is constrained to have the form of a rooted tree. Except for the special entries "." and "..", each directory must appear as an entry in exactly one other directory, which is its parent. The reason for this is to simplify the writing of programs that visit subtrees of the directory structure, and more important, to avoid the separation of portions of the hierarchy. If arbitrary links to directories were permitted, it would be quite difficult to detect when the last connection from the root to a directory was severed.

3.3 Special files

Special files constitute the most unusual feature of the UNIX file system. Each supported I/O device is associated with at least one such file. Special files are read and written just like ordinary disk files, but requests to read or write result in activation of the associated device. An entry for each special file resides in directory **/dev**, although a link may be made to one of these files just as it may to an ordinary file. Thus, for example, to write on a magnetic tape one may write on the file **/dev/mt**. Special files exist for each communication line, each disk, each tape drive, and for physical main memory. Of course, the active disks and the memory special file are protected from indiscriminate access.

There is a threefold advantage in treating I/O devices this way: file and device I/O are as similar as possible; file and device names have the same syntax and meaning, so that a program expecting a file name as a parameter can be passed a device name; finally, special files are subject to the same protection mechanism as regular files.

3.4 Removable file systems

Although the root of the file system is always stored on the same device, it is not necessary that the entire file system hierarchy reside on this device. There is a **mount** system request with two arguments: the name of an existing ordinary file, and the name of a special file whose associated storage volume (e.g., a disk pack) should have the structure of an independent file system containing its own directory hierarchy. The effect of **mount** is to cause references to the heretofore ordinary file to refer instead to the root directory of the file system on the removable volume. In effect, **mount** replaces a leaf of the hierarchy tree (the ordinary file) by a whole new subtree (the hierarchy stored on the removable volume). After the **mount**, there is virtually no distinction between files on the removable volume and those in the permanent file system. In our installation, for example, the root directory resides on a small partition of one of our disk drives, while the other drive, which contains the user's files, is mounted by the system initialization sequence. A mountable file system is generated by writing on its corresponding special file. A utility program is available to create an empty file system, or one may simply copy an existing file system.

There is only one exception to the rule of identical treatment of files on different devices: no link may exist between one file system hierarchy and another. This restriction is enforced so as to avoid the elaborate bookkeeping that would otherwise be required to assure removal of the links whenever the removable volume is dismounted.

3.5 Protection

Although the access control scheme is quite simple, it has some unusual features. Each user of the system is assigned a unique user identification number. When a file is created, it is marked with the user ID of its owner. Also given for new files is a set of ten protection bits. Nine of these specify independently read, write, and execute permission for the owner of the file, for other members of his group, and for all remaining users.

If the tenth bit is on, the system will temporarily change the user identification (hereafter, user ID) of the current user to that of the creator of the file whenever the file is executed as a program. This change in user ID is effective only during the execution of the program that calls for it. The set-user-ID feature provides for privileged programs that may use files inaccessible to other users. For example, a program may keep an accounting file that should neither be read nor changed except by the program itself. If the set-user-ID bit is on for the program, it may access the file although this access might be forbidden to other programs invoked by the given program's user. Since the actual user ID of the invoker of any program is always available, set-user-ID programs may take any measures desired to satisfy themselves as to their invoker's credentials. This mechanism is used to allow users to execute the carefully written commands that call privileged system entries. For example, there is a system entry invokable only by the "super-user" (below) that creates an empty directory. As indicated above, directories are expected to have entries for "." and "..". The command which creates a directory is owned by the super-user and has the set-user-ID bit set. After it checks its invoker's authorization to create the specified directory, it creates it and makes the entries for "." and "..".

Because anyone may set the set-user-ID bit on one of his own files, this mechanism is generally available without administrative intervention. For example, this protection scheme easily solves the MOO accounting problem posed by "Aleph-null."[8]

The system recognizes one particular user ID (that of the "super-user") as exempt from the usual constraints on file access; thus (for example), programs may be written to dump and reload the file system without unwanted interference from the protection system.

3.6 I/O calls

The system calls to do I/O are designed to eliminate the differences between the various devices and styles of access. There is no distinction between "random" and "sequential" I/O, nor is any logical record size imposed by the system. The size of an ordinary file is determined by the number of bytes written on it; no predetermination of the size of a file is necessary or possible.

To illustrate the essentials of I/O, some of the basic calls are summarized below in an anonymous language that will indicate the required parameters without getting into the underlying complexities. Each call to the system may potentially result in an error return, which for simplicity is not represented in the calling sequence.

To read or write a file assumed to exist already, it must be opened by the following call:

```
filep = open ( name, flag )
```

where **name** indicates the name of the file. An arbitrary path name may be given. The **flag** argument indicates whether the file is to be read, written, or "updated," that is, read and written simultaneously.

The returned value **filep** is called a *file descriptor*. It is a small integer used to identify the file in subsequent calls to read, write, or otherwise manipulate the file.

To create a new file or completely rewrite an old one, there is a **create** system call that creates the given file if it does not exist, or truncates it to zero length if it does exist; **create** also opens the new file for writing and, like **open**, returns a file descriptor.

The file system maintains no locks visible to the user, nor is there any restriction on the number of users who may have a file open for reading or writing. Although it is possible for the contents of a file to become scrambled when two users write on it simultaneously, in practice difficulties do not arise. We take the view that locks are neither necessary nor sufficient, in our environment, to prevent interference between users of the same file. They are unnecessary because we are not faced with large, single-file data bases maintained by independent processes. They are insufficient because locks in the ordinary sense, whereby one user is prevented from writing on a file that another user is reading, cannot prevent confusion when, for example, both users are editing a file with an editor that makes a copy of the file being edited.

There are, however, sufficient internal interlocks to maintain the logical consistency of the file system when two users engage simultaneously in activities such as writing on the same file, creating files in the same directory, or deleting each other's open files.

Except as indicated below, reading and writing are sequential. This means that if a particular byte in the file was the last byte written (or read), the next I/O call implicitly refers to the immediately following byte. For each open file there is a pointer, maintained inside the system, that indicates the next byte to be read or written. If *n* bytes are read or written, the pointer advances by *n* bytes.

Once a file is open, the following calls may be used:

```
n = read ( filep, buffer, count )
n = write ( filep, buffer, count )
```

Up to **count** bytes are transmitted between the file specified by **filep** and the byte array specified by **buffer**. The returned value **n** is the number of bytes actually transmitted. In the **write** case, **n** is the same as **count** except under exceptional conditions, such as I/O errors or end of physical medium on special files; in a **read**, however, **n** may without error be less than **count**. If the read pointer is so near the end of the file that reading **count** characters would cause reading beyond the end, only sufficient bytes are transmitted to reach the end of the file; also, typewriter-like terminals never return more than one line of input. When a **read** call returns with **n** equal to zero, the end of the file has been reached. For disk files this occurs when the read pointer becomes equal to the current size of the file. It is possible to generate an end-of-file from a terminal by use of an escape sequence that depends on the device used.

Bytes written affect only those parts of a file implied by the position of the write pointer and the count; no other part of the file is changed. If the last byte lies beyond the end of the file, the file is made to grow as needed.

To do random (direct-access) I/O it is only necessary to move the read or write pointer to the appropriate location in the file.

location = lseek (filep, offset, base)

The pointer associated with **filep** is moved to a position **offset** bytes from the beginning of the file, from the current position of the pointer, or from the end of the file, depending on **base**. **offset** may be negative. For some devices (e.g., paper tape and terminals) seek calls are ignored. The actual offset from the beginning of the file to which the pointer was moved is returned in **location**.

There are several additional system entries having to do with I/O and with the file system that will not be discussed. For example: close a file, get the status of a file, change the protection mode or the owner of a file, create a directory, make a link to an existing file, delete a file.

IV. IMPLEMENTATION OF THE FILE SYSTEM

As mentioned in Section 3.2 above, a directory entry contains only a name for the associated file and a pointer to the file itself. This pointer is an integer called the *i-number* (for index number) of the file. When the file is accessed, its i-number is used as an index into a system table (the *i-list*) stored in a known part of the device on which the directory resides. The entry found thereby (the file's *i-node*) contains the description of the file:

i the user and group-ID of its owner

ii its protection bits

iii the physical disk or tape addresses for the file contents

iv its size

v time of creation, last use, and last modification

vi the number of links to the file, that is, the number of times it appears in a directory

vii a code indicating whether the file is a directory, an ordinary file, or a special file.

The purpose of an **open** or **create** system call is to turn the path name given by the user into an i-number by searching the explicitly or implicitly named directories. Once a file is open, its device, i-number, and read/write pointer are stored in a system table indexed by the file descriptor returned by the **open** or **create**. Thus, during a subsequent call to read or write the file, the descriptor may be easily related to the information necessary to access the file.

When a new file is created, an i-node is allocated for it and a directory entry is made that contains the name of the file and the i-node number. Making a link to an existing file involves creating a directory entry with the new name, copying the i-number from the original file entry, and incrementing the link-count field of the i-node. Removing (deleting) a file is done by decrementing the link-count of the i-node specified by its directory entry and erasing the directory entry. If the link-count drops to 0, any disk blocks in the file are freed and the i-node is de-allocated.

The space on all disks that contain a file system is divided into a number of 512-byte blocks logically addressed from 0 up to a limit that depends on the device. There is space in the i-node of each file for 13 device addresses. For nonspecial files, the first 10 device addresses point at the first 10 blocks of the file. If the file is larger than 10 blocks, the 11 device address points to an indirect block containing up to 128 addresses of additional blocks in the file. Still larger files use the twelfth device address of the i-node to point to a double-indirect block naming 128 indirect blocks, each pointing to 128 blocks of the file. If required, the thirteenth device address is a triple-indirect block. Thus files may conceptually grow to $[(10+128+128^2+128^3)\cdot 512]$ bytes. Once opened, bytes numbered below 5120 can be read with a single disk access; bytes in the range 5120 to 70,656 require two accesses; bytes in the

range 70,656 to 8,459,264 require three accesses; bytes from there to the largest file (1,082,201,088) require four accesses. In practice, a device cache mechanism (see below) proves effective in eliminating most of the indirect fetches.

The foregoing discussion applies to ordinary files. When an I/O request is made to a file whose i-node indicates that it is special, the last 12 device address words are immaterial, and the first specifies an internal *device name*, which is interpreted as a pair of numbers representing, respectively, a device type and subdevice number. The device type indicates which system routine will deal with I/O on that device; the subdevice number selects, for example, a disk drive attached to a particular controller or one of several similar terminal interfaces.

In this environment, the implementation of the **mount** system call (Section 3.4) is quite straightforward. **mount** maintains a system table whose argument is the i-number and device name of the ordinary file specified during the **mount**, and whose corresponding value is the device name of the indicated special file. This table is searched for each i-number/device pair that turns up while a path name is being scanned during an **open** or **create**; if a match is found, the i-number is replaced by the i-number of the root directory and the device name is replaced by the table value.

To the user, both reading and writing of files appear to be synchronous and unbuffered. That is, immediately after return from a **read** call the data are available; conversely, after a **write** the user's workspace may be reused. In fact, the system maintains a rather complicated buffering mechanism that reduces greatly the number of I/O operations required to access a file. Suppose a **write** call is made specifying transmission of a single byte. The system will search its buffers to see whether the affected disk block currently resides in main memory; if not, it will be read in from the device. Then the affected byte is replaced in the buffer and an entry is made in a list of blocks to be written. The return from the **write** call may then take place, although the actual I/O may not be completed until a later time. Conversely, if a single byte is read, the system determines whether the secondary storage block in which the byte is located is already in one of the system's buffers; if so, the byte can be returned immediately. If not, the block is read into a buffer and the byte picked out.

The system recognizes when a program has made accesses to sequential blocks of a file, and asynchronously pre-reads the next block. This significantly reduces the running time of most programs while adding little to system overhead.

A program that reads or writes files in units of 512 bytes has an advantage over a program that reads or writes a single byte at a time, but the gain is not immense; it comes mainly from the avoidance of system overhead. If a program is used rarely or does no great volume of I/O, it may quite reasonably read and write in units as small as it wishes.

The notion of the i-list is an unusual feature of UNIX. In practice, this method of organizing the file system has proved quite reliable and easy to deal with. To the system itself, one of its strengths is the fact that each file has a short, unambiguous name related in a simple way to the protection, addressing, and other information needed to access the file. It also permits a quite simple and rapid algorithm for checking the consistency of a file system, for example, verification that the portions of each device containing useful information and those free to be allocated are disjoint and together exhaust the space on the device. This algorithm is independent of the directory hierarchy, because it need only scan the linearly organized i-list. At the same time the notion of the i-list induces certain peculiarities not found in other file system organizations. For example, there is the question of who is to be charged for the space a file occupies, because all directory entries for a file have equal status. Charging the owner of a file is unfair in general, for one user may create a file, another may link to it, and the first user may delete the file. The first user is still the owner of the file, but it should be charged to the second user. The simplest reasonably fair algorithm seems to be to spread the charges equally among users who have links to a file. Many installations avoid the issue by not charging any fees at all.

V. PROCESSES AND IMAGES

An *image* is a computer execution environment. It includes a memory image, general register values, status of open files, current directory and the like. An image is the current state of a pseudo-computer.

A *process* is the execution of an image. While the processor is executing on behalf of a process, the image must reside in main memory; during the execution of other processes it remains in main memory unless the appearance of an active, higher-priority process forces it to be swapped out to the disk.

The user-memory part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first hardware protection byte boundary above the program text segment in the virtual address space begins a non-shared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the stack pointer fluctuates.

5.1 Processes

Except while the system is bootstrapping itself into operation, a new process can come into existence only by use of the **fork** system call:

```
processid = fork( )
```

When **fork** is executed, the process splits into two independently executing processes. The two processes have independent copies of the original memory image, and share all open files. The new processes differ only in that one is considered the parent process: in the parent, the returned **processid** actually identifies the child process and is never 0, while in the child, the returned value is always 0.

Because the values returned by **fork** in the parent and child process are distinguishable, each process may determine whether it is the parent or child.

5.2 Pipes

Processes may communicate with related processes using the same system **read** and **write** calls that are used for file-system I/O. The call:

```
filep = pipe( )
```

returns a file descriptor **filep** and creates an inter-process channel called a *pipe*. This channel, like other open files, is passed from parent to child process in the image by the **fork** call. A **read** using a pipe file descriptor waits until another process writes using the file descriptor for the same pipe. At this point, data are passed between the images of the two processes. Neither process need know that a pipe, rather than an ordinary file, is involved.

Although inter-process communication via pipes is a quite valuable tool (see Section 6.2), it is not a completely general mechanism, because the pipe must be set up by a common ancestor of the processes involved.

5.3 Execution of programs

Another major system primitive is invoked by

execute (file, arg_1, arg_2, ... , arg_n)

which requests the system to read in and execute the program named by **file**, passing it string arguments arg_1, arg_2, ... , arg_n. All the code and data in the process invoking **execute** is replaced from the **file**, but open files, current directory, and inter-process relationships are unaltered. Only if the call fails, for example because **file** could not be found or because its execute-permission bit was not set, does a return take place from the **execute** primitive; it

resembles a "jump" machine instruction rather than a subroutine call.

5.4 Process synchronization

Another process control system call:

processid = wait (status)

causes its caller to suspend execution until one of its children has completed execution. Then **wait** returns the **processid** of the terminated process. An error return is taken if the calling process has no descendants. Certain status from the child process is also available.

5.5 Termination

Lastly:

exit (status)

terminates a process, destroys its image, closes its open files, and generally obliterates it. The parent is notified through the **wait** primitive, and **status** is made available to it. Processes may also terminate as a result of various illegal actions or user-generated signals (Section VII below).

VI. THE SHELL

For most users, communication with the system is carried on with the aid of a program called the shell. The shell is a command-line interpreter: it reads lines typed by the user and interprets them as requests to execute other programs. (The shell is described fully elsewhere,[9] so this section will discuss only the theory of its operation.) In simplest form, a command line consists of the command name followed by arguments to the command, all separated by spaces:

command arg_1 arg_2 ... arg_n

The shell splits up the command name and the arguments into separate strings. Then a file with name **command** is sought; **command** may be a path name including the "/" character to specify any file in the system. If **command** is found, it is brought into memory and executed. The arguments collected by the shell are accessible to the command. When the command is finished, the shell resumes its own execution, and indicates its readiness to accept another command by typing a prompt character.

If file **command** cannot be found, the shell generally prefixes a string such as **/bin/** to **command** and attempts again to find the file. Directory **/bin** contains commands intended to be generally used. (The sequence of directories to be searched may be changed by user request.)

6.1 Standard I/O

The discussion of I/O in Section III above seems to imply that every file used by a program must be opened or created by the program in order to get a file descriptor for the file. Programs executed by the shell, however, start off with three open files with file descriptors 0, 1, and 2. As such a program begins execution, file 1 is open for writing, and is best understood as the standard output file. Except under circumstances indicated below, this file is the user's terminal. Thus programs that wish to write informative information ordinarily use file descriptor 1. Conversely, file 0 starts off open for reading, and programs that wish to read messages typed by the user read this file.

The shell is able to change the standard assignments of these file descriptors from the user's terminal printer and keyboard. If one of the arguments to a command is prefixed by ">", file descriptor 1 will, for the duration of the command, refer to the file named after the ">". For example:

```
ls
```

ordinarily lists, on the typewriter, the names of the files in the current directory. The command:

```
ls >there
```

creates a file called **there** and places the listing there. Thus the argument **>there** means "place output on **there**." On the other hand:

```
ed
```

ordinarily enters the editor, which takes requests from the user via his keyboard. The command

```
ed <script
```

interprets **script** as a file of editor commands; thus **<script** means "take input from **script**."

Although the file name following "<" or ">" appears to be an argument to the command, in fact it is interpreted completely by the shell and is not passed to the command at all. Thus no special coding to handle I/O redirection is needed within each command; the command need merely use the standard file descriptors 0 and 1 where appropriate.

File descriptor 2 is, like file 1, ordinarily associated with the terminal output stream. When an output-diversion request with ">" is specified, file 2 remains attached to the terminal, so that commands may produce diagnostic messages that do not silently end up in the output file.

6.2 Filters

An extension of the standard I/O notion is used to direct output from one command to the input of another. A sequence of commands separated by vertical bars causes the shell to execute all the commands simultaneously and to arrange that the standard output of each command be delivered to the standard input of the next command in the sequence. Thus in the command line:

```
ls | pr -2 | opr
```

ls lists the names of the files in the current directory; its output is passed to **pr**, which paginates its input with dated headings. (The argument "-2" requests double-column output.) Likewise, the output from **pr** is input to **opr**; this command spools its input onto a file for off-line printing.

This procedure could have been carried out more clumsily by:

```
ls >temp1
pr -2 <temp1 >temp2
opr <temp2
```

followed by removal of the temporary files. In the absence of the ability to redirect output and input, a still clumsier method would have been to require the **ls** command to accept user requests to paginate its output, to print in multi-column format, and to arrange that its output be delivered off-line. Actually it would be surprising, and in fact unwise for efficiency reasons, to expect authors of commands such as **ls** to provide such a wide variety of output options.

A program such as **pr** which copies its standard input to its standard output (with processing) is called a *filter*. Some filters that we have found useful perform character transliteration, selection of lines according to a pattern, sorting of the input, and encryption and decryption.

6.3 Command separators; multitasking

Another feature provided by the shell is relatively straightforward. Commands need not be on different lines; instead they may be separated by semicolons:

```
ls; ed
```

will first list the contents of the current directory, then enter the editor.

A related feature is more interesting. If a command is followed by "**&**," the shell will not wait for the command to finish before prompting again; instead, it is ready immediately to accept a new command. For example:

```
as source >output &
```

causes **source** to be assembled, with diagnostic output going to **output**; no matter how long the assembly takes, the shell returns immediately. When the shell does not wait for the completion of a command, the identification number of the process running that command is printed. This identification may be used to wait for the completion of the command or to terminate it. The "**&**" may be used several times in a line:

```
as source >output & ls >files &
```

does both the assembly and the listing in the background. In these examples, an output file other than the terminal was provided; if this had not been done, the outputs of the various commands would have been intermingled.

The shell also allows parentheses in the above operations. For example:

```
(date; ls) >x &
```

writes the current date and time followed by a list of the current directory onto the file **x**. The shell also returns immediately for another request.

6.4 The shell as a command; command files

The shell is itself a command, and may be called recursively. Suppose file **tryout** contains the lines:

```
as source
mv a.out testprog
testprog
```

The **mv** command causes the file **a.out** to be renamed **testprog**. **a.out** is the (binary) output of the assembler, ready to be executed. Thus if the three lines above were typed on the keyboard, **source** would be assembled, the resulting program renamed **testprog**, and **testprog** executed. When the lines are in **tryout**, the command:

```
sh <tryout
```

would cause the shell **sh** to execute the commands sequentially.

The shell has further capabilities, including the ability to substitute parameters and to construct argument lists from a specified subset of the file names in a directory. It also provides general conditional and looping constructions.

6.5 Implementation of the shell

The outline of the operation of the shell can now be understood. Most of the time, the shell is waiting for the user to type a command. When the newline character ending the line is typed, the shell's **read** call returns. The shell analyzes the command line, putting the arguments in a form appropriate for **execute**. Then **fork** is called. The child process, whose code of course is still that of the shell, attempts to perform an **execute** with the appropriate arguments. If successful, this will bring in and start execution of the program whose name was given. Meanwhile, the other process resulting from the **fork**, which is the parent process, **waits** for the

child process to die. When this happens, the shell knows the command is finished, so it types its prompt and reads the keyboard to obtain another command.

Given this framework, the implementation of background processes is trivial; whenever a command line contains "&," the shell merely refrains from waiting for the process that it created to execute the command.

Happily, all of this mechanism meshes very nicely with the notion of standard input and output files. When a process is created by the **fork** primitive, it inherits not only the memory image of its parent but also all the files currently open in its parent, including those with file descriptors 0, 1, and 2. The shell, of course, uses these files to read command lines and to write its prompts and diagnostics, and in the ordinary case its children—the command programs—inherit them automatically. When an argument with "<" or ">" is given, however, the offspring process, just before it performs **execute**, makes the standard I/O file descriptor (0 or 1, respectively) refer to the named file. This is easy because, by agreement, the smallest unused file descriptor is assigned when a new file is **opened** (or **created**); it is only necessary to close file 0 (or 1) and open the named file. Because the process in which the command program runs simply terminates when it is through, the association between a file specified after "<" or ">" and file descriptor 0 or 1 is ended automatically when the process dies. Therefore the shell need not know the actual names of the files that are its own standard input and output, because it need never reopen them.

Filters are straightforward extensions of standard I/O redirection with pipes used instead of files.

In ordinary circumstances, the main loop of the shell never terminates. (The main loop includes the branch of the return from **fork** belonging to the parent process; that is, the branch that does a **wait**, then reads another command line.) The one thing that causes the shell to terminate is discovering an end-of-file condition on its input file. Thus, when the shell is executed as a command with a given input file, as in:

```
sh <comfile
```

the commands in **comfile** will be executed until the end of **comfile** is reached; then the instance of the shell invoked by **sh** will terminate. Because this shell process is the child of another instance of the shell, the **wait** executed in the latter will return, and another command may then be processed.

6.6 Initialization

The instances of the shell to which users type commands are themselves children of another process. The last step in the initialization of the system is the creation of a single process and the invocation (via **execute**) of a program called **init**. The role of **init** is to create one process for each terminal channel. The various subinstances of **init** open the appropriate terminals for input and output on files 0, 1, and 2, waiting, if necessary, for carrier to be established on dial-up lines. Then a message is typed out requesting that the user log in. When the user types a name or other identification, the appropriate instance of **init** wakes up, receives the log-in line, and reads a password file. If the user's name is found, and if he is able to supply the correct password, **init** changes to the user's default current directory, sets the process's user ID to that of the person logging in, and performs an **execute** of the shell. At this point, the shell is ready to receive commands and the logging-in protocol is complete.

Meanwhile, the mainstream path of **init** (the parent of all the subinstances of itself that will later become shells) does a **wait**. If one of the child processes terminates, either because a shell found an end of file or because a user typed an incorrect name or password, this path of **init** simply recreates the defunct process, which in turn reopens the appropriate input and output files and types another log-in message. Thus a user may log out simply by typing the end-of-file sequence to the shell.

6.7 Other programs as shell

The shell as described above is designed to allow users full access to the facilities of the system, because it will invoke the execution of any program with appropriate protection mode. Sometimes, however, a different interface to the system is desirable, and this feature is easily arranged for.

Recall that after a user has successfully logged in by supplying a name and password, **init** ordinarily invokes the shell to interpret command lines. The user's entry in the password file may contain the name of a program to be invoked after log-in instead of the shell. This program is free to interpret the user's messages in any way it wishes.

For example, the password file entries for users of a secretarial editing system might specify that the editor **ed** is to be used instead of the shell. Thus when users of the editing system log in, they are inside the editor and can begin work immediately; also, they can be prevented from invoking programs not intended for their use. In practice, it has proved desirable to allow a temporary escape from the editor to execute the formatting program and other utilities.

Several of the games (e.g., chess, blackjack, 3D tic-tac-toe) available on the system illustrate a much more severely restricted environment. For each of these, an entry exists in the password file specifying that the appropriate game-playing program is to be invoked instead of the shell. People who log in as a player of one of these games find themselves limited to the game and unable to investigate the (presumably more interesting) offerings of the UNIX system as a whole.

VII. TRAPS

The PDP-11 hardware detects a number of program faults, such as references to nonexistent memory, unimplemented instructions, and odd addresses used where an even address is required. Such faults cause the processor to trap to a system routine. Unless other arrangements have been made, an illegal action causes the system to terminate the process and to write its image on file **core** in the current directory. A debugger can be used to determine the state of the program at the time of the fault.

Programs that are looping, that produce unwanted output, or about which the user has second thoughts may be halted by the use of the **interrupt** signal, which is generated by typing the "delete" character. Unless special action has been taken, this signal simply causes the program to cease execution without producing a **core** file. There is also a **quit** signal used to force an image file to be produced. Thus programs that loop unexpectedly may be halted and the remains inspected without prearrangement.

The hardware-generated faults and the interrupt and quit signals can, by request, be either ignored or caught by a process. For example, the shell ignores quits to prevent a quit from logging the user out. The editor catches interrupts and returns to its command level. This is useful for stopping long printouts without losing work in progress (the editor manipulates a copy of the file it is editing). In systems without floating-point hardware, unimplemented instructions are caught and floating-point instructions are interpreted.

VIII. PERSPECTIVE

Perhaps paradoxically, the success of the UNIX system is largely due to the fact that it was not designed to meet any predefined objectives. The first version was written when one of us (Thompson), dissatisfied with the available computer facilities, discovered a little-used PDP-7 and set out to create a more hospitable environment. This (essentially personal) effort was sufficiently successful to gain the interest of the other author and several colleagues, and later to justify the acquisition of the PDP-11/20, specifically to support a text editing and formatting system. When in turn the 11/20 was outgrown, the system had proved useful enough to persuade management to invest in the PDP-11/45, and later in the PDP-11/70 and Interdata 8/32 machines, upon which it developed to its present form. Our goals throughout the effort, when

articulated at all, have always been to build a comfortable relationship with the machine and to explore ideas and inventions in operating systems and other software. We have not been faced with the need to satisfy someone else's requirements, and for this freedom we are grateful.

Three considerations that influenced the design of UNIX are visible in retrospect.

First: because we are programmers, we naturally designed the system to make it easy to write, test, and run programs. The most important expression of our desire for programming convenience was that the system was arranged for interactive use, even though the original version only supported one user. We believe that a properly designed interactive system is much more productive and satisfying to use than a "batch" system. Moreover, such a system is rather easily adaptable to noninteractive use, while the converse is not true.

Second: there have always been fairly severe size constraints on the system and its software. Given the partially antagonistic desires for reasonable efficiency and expressive power, the size constraint has encouraged not only economy, but also a certain elegance of design. This may be a thinly disguised version of the "salvation through suffering" philosophy, but in our case it worked.

Third: nearly from the start, the system was able to, and did, maintain itself. This fact is more important than it might seem. If designers of a system are forced to use that system, they quickly become aware of its functional and superficial deficiencies and are strongly motivated to correct them before it is too late. Because all source programs were always available and easily modified on-line, we were willing to revise and rewrite the system and its software when new ideas were invented, discovered, or suggested by others.

The aspects of UNIX discussed in this paper exhibit clearly at least the first two of these design considerations. The interface to the file system, for example, is extremely convenient from a programming standpoint. The lowest possible interface level is designed to eliminate distinctions between the various devices and files and between direct and sequential access. No large "access method" routines are required to insulate the programmer from the system calls; in fact, all user programs either call the system directly or use a small library program, less than a page long, that buffers a number of characters and reads or writes them all at once.

Another important aspect of programming convenience is that there are no "control blocks" with a complicated structure partially maintained by and depended on by the file system or other system calls. Generally speaking, the contents of a program's address space are the property of the program, and we have tried to avoid placing restrictions on the data structures within that address space.

Given the requirement that all programs should be usable with any file or device as input or output, it is also desirable to push device-dependent considerations into the operating system itself. The only alternatives seem to be to load, with all programs, routines for dealing with each device, which is expensive in space, or to depend on some means of dynamically linking to the routine appropriate to each device when it is actually needed, which is expensive either in overhead or in hardware.

Likewise, the process-control scheme and the command interface have proved both convenient and efficient. Because the shell operates as an ordinary, swappable user program, it consumes no "wired-down" space in the system proper, and it may be made as powerful as desired at little cost. In particular, given the framework in which the shell executes as a process that spawns other processes to perform commands, the notions of I/O redirection, background processes, command files, and user-selectable system interfaces all become essentially trivial to implement.

Influences

The success of UNIX lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small yet powerful operating system.

The **fork** operation, essentially as we implemented it, was present in the GENIE time-sharing system.[10] On a number of points we were influenced by Multics, which suggested the particular form of the I/O system calls[11] and both the name of the shell and its general functions. The notion that the shell should create a process for each command was also suggested to us by the early design of Multics, although in that system it was later dropped for efficiency reasons. A similar scheme is used by TENEX.[12]

IX. STATISTICS

The following numbers are presented to suggest the scale of the Research UNIX operation. Those of our users not involved in document preparation tend to use the system for program development, especially language work. There are few important "applications" programs.

Overall, we have today:

125	user population
33	maximum simultaneous users
1,630	directories
28,300	files
301,700	512-byte secondary storage blocks used

There is a "background" process that runs at the lowest possible priority; it is used to soak up any idle CPU time. It has been used to produce a million-digit approximation to the constant *e*, and other semi-infinite problems. Not counting this background work, we average daily:

13,500	commands
9.6	CPU hours
230	connect hours
62	different users
240	log-ins

X. ACKNOWLEDGMENTS

The contributors to UNIX are, in the traditional but here especially apposite phrase, too numerous to mention. Certainly, collective salutes are due to our colleagues in the Computing Science Research Center. R. H. Canaday contributed much to the basic design of the file system. We are particularly appreciative of the inventiveness, thoughtful criticism, and constant support of R. Morris, M. D. McIlroy, and J. F. Ossanna.

References

1. L. P. Deutsch and B. W. Lampson, "An online editor," *Comm. Assoc. Comp. Mach.* **10**(12) pp. 793-799, 803 (December 1967).
2. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach.* **18** pp. 151-157 (March 1975).
3. B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, "UNIX Time-Sharing System: Document Preparation," *Bell Sys. Tech. J.* 57(6) pp. 2115-2135 (1978).
4. T. A. Dolotta and J. R. Mashey, "An Introduction to the Programmer's Workbench," *Proc. 2nd Int. Conf. on Software Engineering*, pp. 164-168 (October 13-15, 1976).
5. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "UNIX Time-Sharing System: The Programmer's Workbench," *Bell Sys. Tech. J.* 57(6) pp. 2177-2200 (1978).

6. H. Lycklama, "UNIX Time-Sharing System: UNIX on a Microprocessor," *Bell Sys. Tech. J.* **57**(6) pp. 2087-2101 (1978).

7. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).

8. Aleph-null, "Computer Recreations," *Software Practice and Experience* **1**(2) pp. 201-204 (April-June 1971).

9. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," *Bell Sys. Tech. J.* **57**(6) pp. 1971-1990 (1978).

10. L. P. Deutsch and B. W. Lampson, "SDS 930 time-sharing system preliminary reference manual," Doc. 30.10.10, Project GENIE, Univ. Cal. at Berkeley (April 1965).

11. R. J. Feiertag and E. I. Organick, "The Multics input-output system," *Proc. Third Symposium on Operating Systems Principles*, pp. 35-41 (October 18-20, 1971).

12. D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10," *Comm. Assoc. Comp. Mach.* **15**(3) pp. 135-143 (March 1972).

UNIX For Beginners — Second Edition

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

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

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

Berkeley Notes

This is a standard Bell Laboratories document reproduced without any local modification. Most of the information it contains applies to the UNIX systems on the Berkeley campus, but there are exceptions. This document gives a good general overview of UNIX for people with some previous computer experience. Readers should also investigate sources of local information to learn about Berkeley software, procedures, and policies. Good sources include the online **help** command, the Berkeley edition of the *UNIX Programmers's Manual*, and other documents listed on the *UNIX Documentation Guide* which is available from the Computing Services Library, 218 Evans Hall. Some differences between this document and the Berkeley systems are worth noting:

- The recommended editor at Berkeley is *ex* (and its variants *edit* and *vi*).
- The specific pathnames used in the section "What's in a Filename — Continued" (pp. 7-8) are not the same as those used on the Berkeley systems.
- The default shell, or command line processor, at Berkeley is the C shell (csh), not the standard Bell Laboratories shell (sh). The C shell does not recognize .profile as the name of a login initialization file; instead, it looks for a file called .login. It's treatment of simple commands is the same, but the syntax is different for more complicated things, such as loops. For more information about the C shell, refer to *An Introduction to the C Shell*, by William Joy.

September 29, 1980

†UNIX is a Trademark of Bell Laboratories.

UNIX For Beginners — Second Edition

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

INTRODUCTION

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

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

A word of warning: the UNIX system has become quite popular, and there are several major variants in widespread use. Of course details also change with time. So although the basic structure of UNIX and how to use it is common to all versions, there will certainly be a few things which are different on your system from what is described here. We have tried to minimize the problem, but be aware of it. In cases of doubt, this paper describes Version 7 UNIX.

This paper has five sections:

1. Getting Started: How to log in, how to type, what to do about mistakes in typing, how to log out. Some of this is dependent on which system you log into (phone numbers, for example) and what terminal you use, so this section must necessarily be supplemented by local information.
2. Day-to-day Use: Things you need every day to use the system effectively: generally useful commands; the file system.
3. Document Preparation: Preparing manuscripts is one of the most common uses for UNIX systems. This section contains advice, but not extensive instructions on any of the formatting tools.
4. Writing Programs: UNIX is an excellent system for developing programs. This section talks about some of the tools, but again is not a tutorial in any of the programming languages provided by the system.
5. A UNIX Reading List. An annotated bibliography of documents that new users should be aware of.

I. GETTING STARTED

Logging In

You must have a UNIX login name, which you can get from whoever administers your system. You also need to know the phone number, unless your system uses permanently connected terminals. The UNIX system is capable of dealing with a wide variety of terminals: Terminet 300's; Execuport, TI and similar portables; video (CRT) terminals like the HP2640, etc.; high-priced graphics terminals like the Tektronix 4014; plotting terminals like those from GSI and DASI; and even the venerable Teletype in its various forms. But note: UNIX is strongly oriented towards devices with *lower case*. If your terminal produces only upper case (e.g., model 33 Teletype, some video and portable terminals), life will be so difficult that you should look for another terminal.

Be sure to set the switches appropriately on your device. Switches that might need to be adjusted include the speed, upper/lower case mode, full duplex, even parity, and any others that local wisdom advises. Establish a connection using whatever magic is needed for your terminal; this may involve dialing a telephone call or merely flipping a switch. In either case, UNIX should type "**login:**" at you. If it types garbage, you may be at the wrong speed; check the switches. If that fails, push the "break" or

"interrupt" key a few times, slowly. If that fails to produce a login message, consult a guru.

When you get a **login:** message, type your login name *in lower case.* Follow it by a RETURN; the system will not do anything until you type a RETURN. If a password is required, you will be asked for it, and (if possible) printing will be turned off while you type it. Don't forget RETURN.

The culmination of your login efforts is a "prompt character," a single character that indicates that the system is ready to accept commands from you. The prompt character is usually a dollar sign **$** or a percent sign **%**. (You may also get a message of the day just before the prompt character, or a notification that you have mail.)

Typing Commands

Once you've seen the prompt character, you can type commands, which are requests that the system do something. Try typing

date

followed by RETURN. You should get back something like

Mon Jan 16 14:17:10 EST 1978

Don't forget the RETURN after the command, or nothing will happen. If you think you're being ignored, type a RETURN; something should happen. RETURN won't be mentioned again, but don't forget it — it has to be there at the end of each line.

Another command you might try is **who**, which tells you everyone who is currently logged in:

who

gives something like

mb	**tty01**	**Jan 16**	**09:11**
ski	**tty05**	**Jan 16**	**09:33**
gam	**tty11**	**Jan 16**	**13:07**

The time is when the user logged in; "ttyxx" is the system's idea of what terminal the user is on.

If you make a mistake typing the command name, and refer to a non-existent command, you will be told. For example, if you type

whom

you will be told

whom: not found

Of course, if you inadvertently type the name of some other command, it will run, with more or less mysterious results.

Strange Terminal Behavior

Sometimes you can get into a state where your terminal acts strangely. For example, each letter may be typed twice, or the RETURN may not cause a line feed or a return to the left margin. You can often fix this by logging out and logging back in. Or you can read the description of the command stty in section I of the manual. To get intelligent treatment of tab characters (which are much used in UNIX) if your terminal doesn't have tabs, type the command

stty −tabs

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

Mistakes in Typing

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

dd#atte##e

is the same as **date**.

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

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

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

Read-ahead

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

Stopping a Program

You can stop most programs by typing the character "DEL" (perhaps called "delete" or "rubout" on your terminal). The "interrupt" or "break" key found on most terminals can also be used. In a few programs, like the text editor, DEL stops whatever the program is doing but leaves you in that program. Hanging up the phone will stop most programs.

Logging Out

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

login

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

Mail

When you log in, you may sometimes get the message

You have mail.

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

mail

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

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

mail joe
now type in the text of the letter
on as many lines as you like ...
After the last line of the letter
type the character "control–d",
that is, hold down "control" and type
a letter "d".

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

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

There are other ways to send mail – you can send a previously prepared letter, and you can mail to a number of people all at once. For more details see **mail**(1). (The notation **mail**(1) means the command **mail** in section 1 of the *UNIX Programmer's Manual.*)

Writing to other users

At some point, out of the blue will come a message like

Message from joe tty07...

accompanied by a startling beep. It means that Joe wants to talk to you, but unless you take explicit action you won't be able to talk back. To respond, type the command

write joe

This establishes a two-way communication path. Now whatever Joe types on his terminal will appear on yours and vice versa. The path is slow, rather like talking to the moon. (If you are in the middle of something, you have to get to a state where you can type a command. Normally, whatever program you are running has to terminate or be terminated. If you're editing, you can escape temporarily from the editor – read the editor tutorial.)

A protocol is needed to keep what you type from getting garbled up with what Joe types. Typically it's like this:

Joe types **write smith** and waits.
Smith types **write joe** and waits.
Joe now types his message (as many lines as he likes). When he's ready for a reply, he signals it by typing **(o)**, which stands for "over".
Now Smith types a reply, also terminated by **(o)**.
This cycle repeats until someone gets tired; he then signals his intent to quit with **(oo)**, for "over and out".
To terminate the conversation, each side must type a "control-d" character alone on a line. ("Delete" also works.) When the other person types his "control-d", you will get the message EOF on your terminal.

If you write to someone who isn't logged in, or who doesn't want to be disturbed, you'll be told. If the target is logged in but doesn't answer after a decent interval, simply type "control-d".

On-line Manual

The *UNIX Programmer's Manual* is typically kept on-line. If you get stuck on something, and can't find an expert to assist you, you can print on your terminal some manual section that might help. This is also useful for getting the most up-to-date information on a command. To print a manual section, type "man command-name". Thus to read up on the **who** command, type

man who

and, of course,

man man

tells all about the **man** command.

Computer Aided Instruction

Your UNIX system may have available a program called **learn**, which provides computer aided instruction on the file system and basic commands, the editor, docum nt preparation, and even C programming. Try typing the command

learn

If **learn** exists on your system, it will tell you what to do from there.

II. DAY-TO-DAY USE

Creating Files — The Editor

If you have to type a paper or a letter or a program, how do you get the information stored in the machine? Most of these tasks are done with the UNIX "text editor" **ed**. Since **ed** is thoroughly documented in **ed**(1) and explained in *A Tutorial Introduction to the UNIX Text Editor*, we won't spend any time here describing how to use it. All we want it for right now is to make some *files*. (A file is just a collection of information stored in the machine, a simplistic but adequate definition.)

To create a file called **junk** with some text in it, do the following:

```
ed junk      (invokes the text editor)
a            (command to "ed", to add text)
now type in
whatever text you want ...
.            (signals the end of adding text)
```

The "." that signals the end of adding text must be at the beginning of a line by itself. Don't forget it, for until it is typed, no other **ed** commands will be recognized — everything you type will be treated as text to be added.

At this point you can do various editing operations on the text you typed in, such as correcting spelling mistakes, rearranging paragraphs and the like. Finally, you must write the information you have typed into a file with the editor command **w**:

w

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

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

ed junk

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

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

What files are out there?

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

ls

the response will be

junk
temp

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

ls −t

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

ls −l

will produce something like

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

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

---

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

Options can be combined: ls −lt gives the same thing as ls −l, but sorted into time order. You can also name the files you're interested in, and ls will list the information about them only. More details can be found in ls(1).

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

Printing Files

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

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

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

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

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

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

```
cat junk
```

prints one file, and

```
cat junk temp
```

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

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

```
pr junk temp
```

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

pr can also produce multi-column output:

```
pr −3 junk
```

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

It should be noted that **pr** is *not* a formatting program in the sense of shuffling lines around and justifying margins. The true formatters are **nroff** and **troff**, which we will get to in the section on document preparation.

There are also programs that print files on a high-speed printer. Look in your manual under **opr** and **lpr**. Which to use depends on what equipment is attached to your machine.

Shuffling Files About

Now that you have some files in the file system and some experience in printing them, you can try bigger things. For example, you can move a file from one place to another (which amounts to giving it a new name), like this:

```
mv junk precious
```

This means that what used to be "junk" is now "precious". If you do an ls command now, you will get

```
precious
temp
```

Beware that if you move a file to another one that already exists, the already existing contents are lost forever.

If you want to make a *copy* of a file (that is, to have two versions of something), you can use the **cp** command:

```
cp precious temp1
```

makes a duplicate copy of **precious** in **temp1**.

Finally, when you get tired of creating and moving files, there is a command to remove files from the file system, called **rm**.

```
rm temp temp1
```

will remove both of the files named.

You will get a warning message if one of the named files wasn't there, but otherwise **rm**, like most UNIX commands, does its work silently. There is no prompting or chatter, and error messages are occasionally curt. This terseness is sometimes disconcerting to newcomers, but experienced users find it desirable.

What's in a Filename

So far we have used filenames without ever saying what's a legal name, so it's time for a couple of rules. First, filenames are limited to 14 characters, which is enough to be descriptive.

Second, although you can use almost any character in a filename, common sense says you should stick to ones that are visible, and that you should probably avoid characters that might be used with other meanings. We have already seen, for example, that in the ls command, ls −t means to list in time order. So if you had a file whose name was −t, you would have a tough time listing it by name. Besides the minus sign, there are other characters which have special meaning. To avoid pitfalls, you would do well to use only letters, numbers and the period until you're familiar with the situation.

On to some more positive suggestions. Suppose you're typing a large document like a book. Logically this divides into many small pieces, like chapters and perhaps sections. Physically it must be divided too, for ed will not handle really big files. Thus you should type the document as a number of files. You might have a separate file for each chapter, called

```
chap1
chap2
etc...
```

Or, if each chapter were broken into several files, you might have

```
chap1.1
chap1.2
chap1.3
...
chap2.1
chap2.2
...
```

You can now tell at a glance where a particular file fits into the whole.

There are advantages to a systematic naming convention which are not obvious to the novice UNIX user. What if you wanted to print the whole book? You could say

```
pr chap1.1 chap1.2 chap1.3 .....
```

but you would get tired pretty fast, and would probably even make mistakes. Fortunately, there is a shortcut. You can say

```
pr chap*
```

The * means "anything at all," so this translates into "print all files whose names begin with **chap**", listed in alphabetical order.

This shorthand notation is not a property of the pr command, by the way. It is system-wide, a service of the program that interprets commands (the "shell," sh(1)). Using that fact, you can see how to list the names of the files in the book:

```
ls chap*
```

produces

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

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

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

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

```
pr *
```

prints all your files (alphabetical order), and

```
rm *
```

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

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

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

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

```
pr chap[1−49]*
```

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

The ? pattern matches any single character, so

```
ls ?
```

lists all files which have single-character names, and

```
ls −l chap?.1
```

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

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

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

```
ls '?'
```

We'll see some more examples of this shortly.

What's in a Filename, Continued

When you first made that file called **junk**, how did the system know that there wasn't another **junk** somewhere else, especially since the person in the next office is also reading this tutorial? The answer is that generally each user has a private *directory*, which contains only the files that belong to him. When you log in, you are "in" your directory. Unless you take special action, when you create a new file, it is made in the directory that you are currently in; this is most often your own directory, and thus the file is unrelated to any other file of the same name that might exist in someone else's directory.

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

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

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

/usr/your-name

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

If you now type

ls /usr/your-name

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

Next, try

ls /usr

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

The next step is to try

ls /

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

bin
dev
etc
lib
tmp
usr

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

Now try

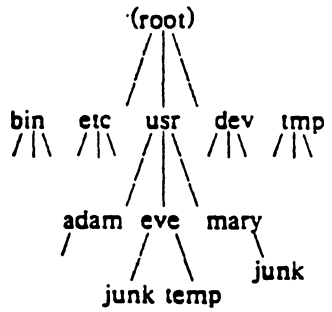
cat /usr/your-name/junk

(if **junk** is still around in your directory). The name

/usr/your-name/junk

is called the **pathname** of the file that you normally think of as "junk". "Pathname" has an obvious meaning: it represents the full name of the path you have to follow from the root through the tree of directories to get to a particular file. It is a universal rule in the UNIX system that anywhere you can use an ordinary filename, you can use a pathname.

Here is a picture which may make this clearer:



Notice that Mary's **junk** is unrelated to Eve's.

This isn't too exciting if all the files of interest are in your own directory, but if you work with someone else or on several projects concurrently, it becomes handy indeed. For example, your friends can print your book by saying

pr /usr/your-name/chap*

Similarly, you can find out what files your neighbor has by saying

ls /usr/neighbor-name

or make your own copy of one of his files by

cp /usr/your-neighbor/his-file yourfile

If your neighbor doesn't want you poking around in his files, or vice versa, privacy can be

arranged. Each file and directory has read-write-execute permissions for the owner, a group, and everyone else, which can be set to control access. See ls(1) and chmod(1) for details. As a matter of observed fact, most users most of the time find openness of more benefit than privacy.

As a final experiment with pathnames, try

```
ls /bin /usr/bin
```

Do some of the names look familiar? When you run a program, by typing its name after the prompt character, the system simply looks for a file of that name. It normally looks first in your directory (where it typically doesn't find it), then in **/bin** and finally in **/usr/bin**. There is nothing magic about commands like **cat** or **ls**, except that they have been collected into a couple of places to be easy to find and administer.

What if you work regularly with someone else on common information in his directory? You could just log in as your friend each time you want to, but you can also say "I want to work on his files instead of my own". This is done by changing the directory that you are currently in:

```
cd /usr/your-friend
```

(On some systems, **cd** is spelled **chdir**.) Now when you use a filename in something like **cat** or **pr**, it refers to the file in your friend's directory. Changing directories doesn't affect any permissions associated with a file — if you couldn't access a file from your own directory, changing to another directory won't alter that fact. Of course, if you forget what directory you're in, type

```
pwd
```

to find out.

It is usually convenient to arrange your own files so that all the files related to one thing are in a directory separate from other projects. For example, when you write your book, you might want to keep all the text in a directory called **book**. So make one with

```
mkdir book
```

then go to it with

```
cd book
```

then start typing chapters. The book is now found in (presumably)

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

To remove the directory **book**, type

```
rm book/*
rmdir book
```

The first command removes all files from the directory; the second removes the empty directory.

You can go up one level in the tree of files by saying

```
cd ..
```

".." is the name of the parent of whatever directory you are currently in. For completeness, "." is an alternate name for the directory you are in.

Using Files instead of the Terminal

Most of the commands we have seen so far produce output on the terminal; some, like the editor, also take their input from the terminal. It is universal in UNIX systems that the terminal can be replaced by a file for either or both of input and output. As one example,

```
ls
```

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

```
ls >filelist
```

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

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

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

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

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

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

```
ed file <script
```

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

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

Pipes

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

For example,

```
pr f g h
```

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

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

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

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

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

There are many other examples of pipes. For example,

```
ls | pr -3
```

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

```
who | wc
```

tells how many people are logged on. And of course

```
ls | wc
```

counts your files.

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

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

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

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

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

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

The Shell

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

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

```
date; who
```

does both commands before returning with a prompt character.

You can also have more than one program running *simultaneously* if you wish. For example, if you are doing something time-consuming, like the editor script of an earlier section, and you don't want to wait around for the results before starting something else, you can say

```
ed file <script &
```

The ampersand at the end of a command line says "start this command running, then take further commands from the terminal immediately," that is, don't wait for it to complete. Thus the script will begin, but you can do something else at the same time. Of course, to keep the output from interfering with what you're doing on the terminal, it would be better to say

```
ed file <script >script.out &
```

which saves the output lines in a file called **script.out**.

When you initiate a command with &, the system replies with a number called the process number, which identifies the command in case you later want to stop it. If you do, you can say

```
kill process-number
```

If you forget the process number, the command **ps** will tell you about everything you have running. (If you are desperate, **kill 0** will kill all your processes.) And if you're curious about other people, **ps a** will tell you about *all* programs that are currently running.

You can say

```
(command-1; command-2; command-3) &
```

to start three commands in the background, or you can start a background pipeline with

```
command-1 | command-2 &
```

Just as you can tell the editor or some simi-

lar program to take its input from a file instead of from the terminal, you can tell the shell to read a file to get commands. (Why not? The shell, after all, is just a program, albeit a clever one.) For instance, suppose you want to set tabs on your terminal, and find out the date and who's on the system every time you log in. Then you can put the three necessary commands (**tabs**, **date**, **who**) into a file, let's call it **startup**, and then run it with

sh startup

This says to run the shell with the file **startup** as input. The effect is as if you had typed the contents of **startup** on the terminal.

If this is to be a regular thing, you can eliminate the need to type **sh**: simply type, once only, the command

chmod +x startup

and thereafter you need only say

startup

to run the sequence of commands. The **chmod**(1) command marks the file executable; the shell recognizes this and runs it as a sequence of commands.

If you want **startup** to run automatically every time you log in, create a file in your login directory called **.profile**, and place in it the line **startup**. When the shell first gains control when you log in, it looks for the **.profile** file and does whatever commands it finds in it. We'll get back to the shell in the section on programming.

III. DOCUMENT PREPARATION

UNIX systems are used extensively for document preparation. There are two major formatting programs, that is, programs that produce a text with justified right margins, automatic page numbering and titling, automatic hyphenation, and the like. **nroff** is designed to produce output on terminals and line-printers. **troff** (pronounced "tee-roff") instead drives a phototypesetter, which produces very high quality output on photographic paper. This paper was formatted with **troff**.

Formatting Packages

The basic idea of **nroff** and **troff** is that the text to be formatted contains within it "formatting commands" that indicate in detail how the formatted text is to look. For example, there might be commands that specify how long lines are, whether to use single or double spacing, and what running titles to use on each page.

Because **nroff** and **troff** are relatively hard to learn to use effectively, several "packages" of canned formatting requests are available to let you specify paragraphs, running titles, footnotes, multi-column output, and so on, with little effort and without having to learn **nroff** and **troff**. These packages take a modest effort to learn, but the rewards for using them are so great that it is time well spent.

In this section, we will provide a hasty look at the "manuscript" package known as **−ms**. Formatting requests typically consist of a period and two upper-case letters, such as **.TL**, which is used to introduce a title, or **.PP** to begin a new paragraph.

A document is typed so it looks something like this:

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

The lines that begin with a period are the formatting requests. For example, **.PP** calls for starting a new paragraph. The precise meaning of **.PP** depends on what output device is being used (typesetter or terminal, for instance), and on what publication the document will appear in. For example, **−ms** normally assumes that a paragraph is preceded by a space (one line in **nroff**, ½ line in **troff**), and the first word is indented. These rules can be changed if you like, but they are changed by changing the interpretation of **.PP**, not by re-typing the document.

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

troff −ms files ...

for the typesetter, and

nroff −ms files ...

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

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

Supporting Tools

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

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

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

produces the output

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

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

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

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

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

```
grep 'ing$' chap*
```

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

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

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

```
tr A-Z a-z <input >output
```

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

Most of these programs are either independently documented (like **eqn** and **tbl**), or are sufficiently simple that the description in the *UNIX Programmer's Manual* is adequate explanation.

Hints for Preparing Documents

Most documents go through several versions (always more than you expected) before they are finally finished. Accordingly, you should do whatever possible to make the job of changing them easy.

First, when you do the purely mechanical operations of typing, type so that subsequent editing will be easy. Start each sentence on a new line. Make lines short, and break lines at natural places, such as after commas and semicolons, rather than randomly. Since most people change documents by rewriting phrases and adding, deleting and rearranging sentences, these precautions simplify any editing you have to do later.

Keep the individual files of a document down to modest size, perhaps ten to fifteen thousand characters. Larger files edit more slowly, and of course if you make a dumb mistake it's better to have clobbered a small file than a big one. Split into files at natural boundaries in the document, for the same reasons that you start each sentence on a new line.

The second aspect of making change easy is to not commit yourself to formatting details too early. One of the advantages of formatting packages like **-ms** is that they permit you to delay decisions to the last possible moment. Indeed, until a document is printed, it is not even decided whether it will be typeset or put on a line printer.

As a rule of thumb, for all but the most trivial jobs, you should type a document in terms of a set of requests like .PP, and then define them appropriately, either by using one of the canned packages (the better way) or by defining your own **nroff** and **troff** commands. As long as you have entered the text in some systematic way, it can always be cleaned up and reformatted by a judicious combination of editing commands and request definitions.

IV. PROGRAMMING

There will be no attempt made to teach any of the programming languages available but a few words of advice are in order. One of the reasons why the UNIX system is a productive programming environment is that there is already a rich set of tools available, and facilities like pipes, I/O redirection, and the capabilities of the shell often make it possible to do a job by pasting together programs that already exist instead of writing from scratch.

The Shell

The pipe mechanism lets you fabricate quite complicated operations out of spare parts that already exist. For example, the first draft of the **spell** program was (roughly)

```
cat ...     collect the files
| tr ...    put each word on a new line
| tr ...    delete punctuation, etc.
| sort      into dictionary order
| uniq      discard duplicates
| comm      print words in text
              but not in dictionary
```

More pieces have been added subsequently, but this goes a long way for such a small effort.

The editor can be made to do things that would normally require special programs on other systems. For example, to list the first and last lines of each of a set of files, such as a book, you could laboriously type

```
ed
e chap1.1
1p
$p
e chap1.2
1p
$p
etc.
```

But you can do the job much more easily. One way is to type

```
ls chap* >temp
```

to get the list of filenames into a file. Then edit this file to make the necessary series of editing commands (using the global commands of **ed**), and write it into **script**. Now the command

```
ed <script
```

will produce the same output as the laborious hand typing. Alternately (and more easily), you can use the fact that the shell will perform loops, repeating a set of commands over and over again for a set of arguments:

```
for i in chap*
do
        ed $i <script
done
```

This sets the shell variable i to each file name in turn, then does the command. You can type this command at the terminal, or put it in a file for later execution.

Programming the Shell

An option often overlooked by newcomers is that the shell is itself a programming language, with variables, control flow (**if-else**, **while**, **for**, **case**), subroutines, and interrupt handling. Since there are many building-block programs, you can sometimes avoid writing a new program merely by piecing together some of the building blocks with shell command files.

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

Programming in C

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

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

C programs that don't depend too much on special features of UNIX (such as pipes) can be moved to other computers that have C compilers. The list of such machines grows daily; in addition to the original PDP-11, it currently

includes at least Honeywell 6000, IBM 370, Interdata 8/32, Data General Nova and Eclipse, HP 2100, Harris /7, VAX 11/780, SEL 86, and Zilog Z80. Calls to the standard I/O library will work on all of these machines.

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

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

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

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

Other Languages

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

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

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

V. UNIX READING LIST

General:

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

Documents for Use with the UNIX Time-sharing System. Volume 2 of the Programmer's Manual. This contains more extensive descriptions of major commands, and tutorials and reference manuals. All of the papers listed below are in it, as are descriptions of most of the programs mentioned above.

D. M. Ritchie and K. L. Thompson, "The UNIX Time-sharing System," CACM, July 1974. An overview of the system, for people interested in operating systems. Worth reading by anyone who programs. Contains a remarkable number of one-sentence observations on how to do things right.

The Bell System Technical Journal (BSTJ) Special Issue on UNIX, July/August, 1978, contains many papers describing recent developments, and some retrospective material.

The 2nd International Conference on Software Engineering (October, 1976) contains several papers describing the use of the Programmer's Workbench (PWB) version of UNIX.

Document Preparation:

B. W. Kernighan, "A Tutorial Introduction to the UNIX Text Editor" and "Advanced Editing on UNIX," Bell Laboratories, 1978. Beginners need the introduction; the advanced material will help you get the most out of the editor.

M. E. Lesk, "Typing Documents on UNIX," Bell Laboratories, 1978. Describes the **-ms** macro package, which isolates the novice from the vagaries of **nroff** and **troff**, and takes care of

most formatting situations. If this specific package isn't available on your system, something similar probably is. The most likely alternative is the PWB/UNIX macro package **−mm**; see your local guru if you use PWB/UNIX.

B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," Bell Laboratories Computing Science Tech. Rep. 17.

M. E. Lesk, "Tbl — A Program to Format Tables," Bell Laboratories CSTR 49, 1976.

J. F. Ossanna, Jr., "NROFF/TROFF User's Manual," Bell Laboratories CSTR 54, 1976. **troff** is the basic formatter used by **−ms**, **eqn** and **tbl**. The reference manual is indispensable if you are going to write or maintain these or similar programs. But start with:

B. W. Kernighan, "A TROFF Tutorial," Bell Laboratories, 1976. An attempt to unravel the intricacies of **troff**.

Programming:

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978. Contains a tutorial introduction, complete discussions of all language features, and the reference manual.

B. W. Kernighan and D. M. Ritchie, "UNIX Programming," Bell Laboratories, 1978. Describes how to interface with the system from C programs: I/O calls, signals, processes.

S. R. Bourne, "An Introduction to the UNIX Shell," Bell Laboratories, 1978. An introduction and reference manual for the Version 7 shell. Mandatory reading if you intend to make effective use of the programming power of this shell.

S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Bell Laboratories CSTR 32, 1978.

M. E. Lesk, "Lex — A Lexical Analyzer Generator," Bell Laboratories CSTR 39, 1975.

S. C. Johnson, "Lint, a C Program Checker," Bell Laboratories CSTR 65, 1977.

S. I. Feldman, "MAKE — A Program for Maintaining Computer Programs," Bell Laboratories CSTR 57, 1977.

J. F. Maranzano and S. R. Bourne, "A Tutorial Introduction to ADB," Bell Laboratories CSTR 62, 1977. An introduction to a powerful but complex debugging tool.

S. I. Feldman and P. J. Weinberger, "A Portable Fortran 77 Compiler," Bell Laboratories, 1978. A full Fortran 77 for UNIX systems.

Communicating with UNIX†

Ricki Blau
Computing Services
University of California
Berkeley, California 94720

ABSTRACT

This narrative introduction to the use of the UNIX system assumes no prior familiarity with computers. Its aim is to lead the beginning user through the first few sessions with UNIX. It starts with the use of the terminal and the login procedure, and later presents the fundamental system features and commands. Introductions to the file system and command line interpreter are given.

September 1981

†UNIX is a trademark of Bell Laboratories.

Contents

Session 1: Introduction

UNIX is an interactive computer system — people and the system talk back and forth with each other by means of a terminal connected to the computer. When users give instructions to UNIX, they are communicating with a program, a set of instructions which has been given to the computer telling it how to perform some task. UNIX is a collection of many programs, collectively called the UNIX system, which monitors the use of the machinery and supervises all of the programs that make up the system.

This is the first in a series of tutorials to introduce you to UNIX. Don't hesitate to experiment while you are becoming familiar with UNIX. An interactive system responds quickly to instructions, informing you about the outcome of each task and asking for a new command as soon as it is ready to receive one. If you enter a command which doesn't work the way you had expected, make a change and try again. Learning about the system by using it is the best way to become familiar with UNIX.

Your account

On the Berkeley campus there are several nearly identical, but separate, computers on which the UNIX system is available to University personnel and students. All of the UNIX systems are very much the same, but the computers aren't completely interchangeable; so, each computer is assigned a name to differentiate it from the other computers. Thus when we refer to UNIX A, or UNIX E, we are referring to the system on a particular computer, in our example, the A machine or the E machine.

Each account is assigned to one of the campus UNIX systems. To use your account, you must make contact with the proper system. When you are given your account, by either the Computing Services Accounting office or by the instructor of your class, you will be told the name of the UNIX system you must use and how to access it.

Associated with your account are two pieces of identification. The *login name* is the name by which your account is known.† Your *password* keeps unauthorized people from using your account. Whenever you want to use UNIX, you will need to establish your identity by typing both your login name and password.

The terminal

Entering information at a terminal is very much like typing on an electric typewriter. Terminals display information either on paper (*printing* or *hardcopy* devices) or through light displays on a tv screen (*CRT* terminals). Whether a terminal writes on paper or on a CRT screen, it interacts with the computer in the same way, and you may switch from one device to another at your convenience. Terminals vary in their characteristics from model to model, and the labelling and placement of some keys does vary. When it is likely that devices may differ, alternative suggestions will be given for using different terminals. It may be necessary to test a few of the alternatives to find out which is appropriate for the equipment you are using.

The keyboard

Examine your terminal — most of the keys are the same as on a standard typewriter keyboard. Notice that there is always a key for the number one, "1". On a terminal the number one ("1") and the lower-case letter "ell" ("l") are **not** interchangeable. When you type a lower-case "l" (ell) the computer always interprets it as the letter "l" (ell) and never as the

† The login names for all of the students in a class usually start with the course number (and possibly the section name) followed by a different pair of letters for each student.

number "1" (one). Another character that looks similar to these but is distinct from both is the vertical bar "|", which is the shift-\ on the keyboard. Similarly, the letter "O" and the number "0" (zero) cannot be interchanged.

Also, a blank is actually a specific character to UNIX. When you press the space bar (located at the bottom of the keyboard, just as on a typewriter) you are actually instructing UNIX to perceive a blank space.

The RETURN key

Besides the normal typewriter keys, there are some special-purpose keys you will be using frequently. The RETURN, marked "RETURN" or "RET", is usually located near the upper right corner of the keyboard. As commands are typed on a terminal, they are read by UNIX character-by-character as they are entered. The command is examined and executed only after the RETURN key is pressed. This serves as a signal to the system that you have finished entering a line and that it is UNIX's turn to handle the information it has received. In these lessons, the symbol *<cr>* is written to indicate that the RETURN key should be pressed.

Suppose you wanted to enter a particularly lengthy command but found yourself at the terminal's right hand margin before you were done. By simply continuing to type, you may on some terminals enter the command, even though it may be longer than one physical line. These terminals will automatically advance to give you a fresh line to type on. When you are ready for UNIX to respond to what you have typed, press RETURN. On many terminals, the key that may be labelled either "NEW LINE" or "LINE FEED" can be used just as the RETURN. If UNIX doesn't respond when you think it should, try typing a RETURN or its equivalent. If that doesn't work you will need to examine your most recent commands or lines of text to determine what UNIX thinks you are doing. If that doesn't help, you will need to seek other advice.

The control key

The control key is similar in many ways to the shift key. Just as the shift key allows the other keys to have a second meaning, the control key (usually marked "CTRL") gives many keys a third meaning. To use the control key, hold down CTRL while striking a second key. The characters you type when you hold down CTRL and press another key are invisible — they have meanings to UNIX, but they don't correspond to any character that can be printed or displayed. Some of the control characters are used to give useful instructions to the terminal. These invisible characters are just as real to UNIX as any characters that can appear on the terminal. If UNIX ever has trouble recognizing a command or a name that looks perfectly good on the terminal, the problem may be due to invisible control characters that crept into a word through your typing error.

Some programs show you where you have typed a control character by printing a representation of the control character in symbolic form. Most of these programs follow the convention of printing a "^" to mean "control" followed by the letter that was typed. On some terminals, the "^" appears as a "↑". For example, if a program prints that you have typed c^At, it means that the character control-A appeared between the "c" and the "t". The control-A would have been entered by your holding down the control key while typing an "A", and on many terminal keyboards can be generated if your finger hits between the CTRL and "A" by accident. We will discuss some specific control characters when the time comes to use them. Now we should be ready to make contact with UNIX.

Connecting

If your terminal is not already on, now's the time to turn it on. On a typewriter terminal, the ON–OFF switch is usually clearly marked on the front of the device. On ADM terminals (the model name of most of the public CRT terminals), there is a toggle switch on the back of the device on the lower right hand corner; press the switch toward your right.

Port selector terminals

If your terminal has a small box attached with a red button on top, it is connected to the port selector. You can request any of several UNIX systems by turning on the terminal and pressing the red button once. The terminal will respond with

Request:

Type the name of the system you are to use. For this example, we use the Computer Facilities and Operations (C F & O) system E:

Request:e *(and press RETURN)*

The terminal will skip a line, emit a beep, and then print a greeting inviting you to login. You are now ready to login.

Bussiplexer terminals

The bussiplexer is a communications network that makes it possible for you to connect to any of several C F & O UNIX systems. Use the same techniques as above to turn on the terminal or to connect over dial-up terminals, as described below. If you don't immediately receive a login message, press RETURN. The login message will be similar to:

U.C. Bussiplexer (A+B+C+D+E+F+)

:login:

It lists the systems that are connected to the bussiplexer. A "+" or "−" follows each name. A "+" indicates that the system is up (in service) and a "−" means that the system is down (unavailable).

After the message of greeting appears, type your login name and press RETURN. Frequently the characters you enter are slow to print or appear on the screen. Input is ignored for a second or so after the message of greeting is printed. A good strategy is to type one character and wait a few seconds for it to appear. If it doesn't, try again. Once your first character appears, you may type the rest of your login name, followed by a <*cr*>.

For most accounts the bussiplexer knows the system your account is on, and so it will respond:

Connecting to Unix E

if E is the UNIX system your account is on. If the bussiplexer does not know which of the systems has your account, you will be asked to select one. Type a single letter for the system of your choice (here, "e" because our example account is on UNIX E) followed by a RETURN:

Select Host: e<*cr*>
Connecting to Unix E

Connected

After a few seconds, your UNIX system will ask for a password:

Password: *(type your password and press RETURN)*

You are now logged in.

Directly-linked terminals

Turn on your terminal and press the RETURN key. You are now ready to login.

Dial-up terminals

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. You are now ready to login.†

Logging in

The message inviting you to login is:

:login:

Type your login name, which identifies you to UNIX, on the same line as the login message, and press RETURN. If you make a mistake while typing your login name simply press the "@" key, which tells UNIX to ignore the line you have typed so far and lets you begin again as though you had typed nothing at all. 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 "sherlock":

:login: **sherlock** *(and press the RETURN key)*

(In the examples, input you would type appears in "**boldface**" 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 RETURN)*

As with typing your login name, if you think you have made a mistake you can type the @ character to tell UNIX to ignore what you have typed on the line so far, and to take what you type next as your password. 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. Once UNIX accepts the password, you are logged into the system. UNIX will print the message of the day , such as

For latest news type 'help news'

Erase set to control-H

The last line, "Erase set to control-H", indicates that you can correct typing errors in the line you are typing by holding down the CTRL key and typing the "H" key. If you try typing control-H, you will notice that the terminal backspaces in the line you are on. You can backspace over your error, and then type what you want to be the rest of the line.

Prompting

The greeting message, which may consist of a few or many lines, will be followed by a prompt from UNIX, the percent sign "%". The prompt is how UNIX indicates it is ready to receive commands. After each command has been executed, UNIX responds with a new prompt to let you know that it expects another command.

---

†If your terminal prints two letters for every one you type, your terminal is set for "half duplex," which is used when communicating with the IBM system. Set the duplex switch on your terminal and coupler to "full duplex." Also, some terminals and acoustic couplers have a "local copy" switch that you should set to "out" to prevent each character being printed twice.

A summary

Up to now, the exchange with UNIX should look something like this:

C F & O Version 7 Unix System — "E"

:login: **sherlock**<*cr*>
Password:<*cr*>

For latest news type 'help news'

Erase set to control-H

%

Try typing **help news**, followed by RETURN, and the latest system announcements will appear:

% **help news**<*cr*>
U N I X N E W S

and the rest of the news will follow. If you're working at a CRT terminal, only one screenful of news will be sent to your terminal at a time. At the bottom of the page will be the instruction:

--More--[Press space to continue, Rubout to abort]

Pressing the space bar is the way to ask the *help* program for the next screenful of information when you are ready to read it. If you press RETURN instead of the space bar, the system will respond with a line of text and repeat the above message until you hit the space bar or the RUB key.

Interrupting command execution

The key labelled either "RUB" (for RUBOUT) or "DEL" (for DELETE) is another special purpose key. If you issue a command, and wish to stop its execution before it stops of its own accord, press this key. On ADM terminals, you will have to shift in order to type "RUB". On some terminals the BREAK key will function as RUB (DELETE).

For the time being, stop the transmission of news to your terminal by pressing the RUB key. Even though a typewriter terminal may not pause after a "page" of print, you can also stop the transmission of the news by pressing RUB or its equivalent. When you have successfully stopped the printing of news, UNIX will respond with a new prompt ("%").

Sometimes it is desirable to slow down or temporarily interrupt transmission to your terminal, even though you do not want to stop the execution of a command permanently. The special character "control-S", typed by holding down the CTRL key and then also pressing "S", can be used to interrupt printing. Type control-S, and the screen will freeze; when you type "control-Q", by holding down the CTRL key and pressing "Q", you will restart the transmission to your terminal. The control-S/control-Q sequence can also be used to pause and restart at a printing terminal. If you'd like to see the rest of the announcements, ask for the news again once the system is ready to receive new commands. Use control-S and control-Q to see some of the news a few lines at a time.

Help

The **help** command is set up to provide announcements and information about many aspects of the system. To find out what sort of information is available, type **help** followed by RETURN.

% **help**<*cr*>

When you type the command **help**, UNIX looks for a program named *help* that has the necessary instructions to send information to your terminal. If the command is mistyped, such as happens when a key "bounces" (repeats itself):

% **helppp**< *cr*>

UNIX will look for a program called *helppp*. Mistype this command deliberately, and you will see how UNIX responds. There is no program by the exact name *helppp* so the system tells you:

```
% helppp< cr>
helppp: Command not found.
%
```

The message is followed by a new prompt, so you can start again and enter the command correctly. You might want to try some of the commands described by *help*.

Logging out

When you decide to leave UNIX for the time being, you will have to log out. This is done by typing the command

% **logout**

It is not sufficient simply to turn off the terminal; you will remain logged in until you type the logout command.

It is very important to remember to log off. If you should leave the terminal before logging off, any unscrupulous or unwitting stranger who sits down at the terminal after you leave could continue to use your account.

This is the end of the first session with UNIX.

Session 2: Simple Commands

Login with UNIX as in the first session. Once you are connected to the system, UNIX will print various messages and then signal its readiness to accept commands with the percent-sign prompt ("%"). (The examples will no longer show the RETURN key that must be typed at the end of each line to let the system know that the line is complete.)

```
C F & O Version 7 Unix System — "E"

:login: sherlock
Password: (type the password)
Last login: Wed Sep 10 09:41:29 on bx077

For latest news type 'help news'

Erase set to control-H
%
```

To review, communicating with UNIX consists of your giving input to programs which tell the computer machinery how to respond to commands. For instance, there is a program which keeps track of the terminals which are connected to UNIX. If a terminal is turned on and no one is using it, this program prints the ":login:" message. When a name and password are given correctly, it lets the user onto the system. Once this process has been completed, another program is called in to supervise the rest of the session with UNIX.

The shell

It is this second program, the "shell", that coordinates communication between your terminal and UNIX. It reads commands and directs them, like a conductor in an orchestra, to other programs that will actually execute the commands.

The prompt signal ("%") comes from the shell. If you should respond by typing **date**, the system reads the letters and stores them in a temporary place. Once you indicate that the command is finished by pressing RETURN, UNIX can respond:

```
% date
Wed Sep 24 09:55:14 PDT 1981
%
```

When you type **date**, the shell looks for a program called *date* and directs it to execute. The date program then executes its task, that of typing the date and time on the terminal.

Making corrections

As we found in Session 1, UNIX provides a convenient way to correct typing errors that you notice while you are still typing a line. By holding the CTRL key and typing the "H" key, the terminal responds by backspacing one space. We will represent control-H by "^H". To review, the following lines are equivalent:

```
% help news

% helppp^H^H news
```

and, recalling that a blank space is actually a character to a computer, so is

```
% helpp n^H^H^H news
```

Remember that "^H" means that you hold the CTRL key as you would the SHIFT key, and press the key labelled "H". It is necessary to erase both the "n" and the blank preceding it before erasing the extra "p", so three control-H characters were used.

Control-H can be used to erase all the way back to the beginning of a line. Suppose, however, that a mistake that you notice is so far behind your current position that it doesn't seem

worthwhile to try to salvage the line. Remember that the at-sign ("@") will erase the contents of an entire line up to the "@", if it is typed before the RETURN key is pressed, and you will be given a fresh line. (If you are on a bussiplexer terminal, you will not advance to the next line after hitting "@", but there is no other difference in the way the at-sign is handled.)

```
% hlep ne@
help news
```

Begin the line again immediately after the at-sign, and when the corrected line is complete, press RETURN. Erasing letters or lines with "^H" or "@" must be done *before* pressing RETURN. Once you have pressed RETURN, whatever you typed, including any mistakes not erased by "^H" or "@", will be evaluated by the shell.

Changing the password

If the original password for your account was assigned randomly by your instructor or the Computing Services Accounting office, one of the first things you should do is reset it to something you've chosen yourself.

```
% passwd sherlock
```

Use the above command, substituting your own login name for **sherlock.** You will first be asked to type your current password, which will not appear on the screen. Remember to press the RETURN key after the password. Next you will be asked twice to enter the new password. The purpose of repeating the password is to minimize the chance of typing errors.

```
% passwd sherlock
Old password: (type your current password, followed by < cr> )
New password: (type the new password, followed by < cr> )
Retype new password: (repeat the new password and the < cr> )
```

Longer passwords are harder to guess, and therefore protect your account more securely. The *passwd* program will ask you to use a longer password if the one you first enter is too simple. It's best to use passwords that are at least six characters long. Passwords may be as long as eight characters and may contain any characters but "^H" and "@".

If you should forget the password and are prevented from logging in, the problem can be corrected. If you have a class account, contact the instructor or TA. Other users can submit a request at the Accounting office, 239 Evans Hall, to have a new password entered for an account.

Sending mail

You can use UNIX to send mail to any user whose login name you know. The format of the command is the word **mail** followed by the login name of the user who is to receive mail. Follow the command with a RETURN key, and then start writing your message, using the RETURN key whenever you need a new line.

```
% mail sherlock
Whatever is typed on the lines following the
mail command will be reproduced as
mail in sherlock's UNIX mail box.
Finish off the message by typing
a control-D on a line by itself.
% (a control-D was typed on this line, then the system typed back the "%" )
```

Send some mail to yourself to see how it works. When you have typed the last line of the mail, return to a new line and type a control-D. You will be informed about the presence of mail the next time you login.

Another command, **trouble**, works very much like **mail**. It automatically sends a report of trouble to the proper people, and can be used, for example, to report terminal malfunctions. You can find out more about it by typing

```
% help trouble
```

Receiving mail

Log out after you finish sending mail, and then log back in. When you login again, there will be a message for you:

```
You have mail.
% mail
```

Retrieve your mail with the command **mail** alone on a line followed by RETURN. The **mail** command will print any messages waiting in your mail box. After each message, **mail** prints a prompt of "?" to ask you what you want to do with the message. If you type a "d" (for "delete") followed by RETURN, the message will disappear. The command "s" (for "save") asks mail to save a copy of the message in a file called *mbox*. More pieces of mail may be added to *mbox* even if others are already stored there. You can type "+" to go on to the next message. With a "q", you can quit reading your mail. Any unread messages will be waiting for you the next time you use the command **mail** to read your mail. If you need to be reminded about the different **mail** commands, or if you'd like to find out about further commands, you can type a "?" in response to **mail**'s prompt and receive a list of commands.

Save the mail you have sent to yourself to be retrieved in a later session. Although you are only notified about mail upon login, the command **mail** may be used at any time during a UNIX session to see any mail that may be waiting for you. There should no longer be any mail for you, but type **mail** anyhow to see what happens. Finish the session by typing **logout**.

This is the end of the second session. There is a similar series, *Edit: A Tutorial*, which show how to use the text editor to create files of text. Before starting the next session of *Communicating with UNIX* you should be familiar with the material in the first two sessions of the edit tutorial.

Session 3: Files

By now you should be familiar with using the text editor to create files. It's assumed that you've already begun a tutorial that explains the use of the editor, such as *Edit: A Tutorial*, available from the Computing Services Library, 218 Evans Hall. We'll review a few terms before elaborating on the UNIX file system.

Files

A file is a logical unit of data that is stored on a computer system. For example, the contents of a file might consist of a program, the text of a paper, or the data for a program. Once you have created a file, it is stored for you until you instruct the system to remove it. You may create a file during one UNIX session, log off, and return to use it at a later time. Files are stored on a device called a disk, which looks like a stack of phonograph records. Each surface is covered with a material similar to the coating on magnetic recording tape, on which the data is written.

Files can contain anything you write and store in them, from one small number to the text of a thesis. Keeping files organized is usually easiest when some logical unit of data, such as a program or a chapter of a book, is stored in each file.

Filenames

Filenames serve the same purpose as the labels of manila folders in a file cabinet. They are used to distinguish one set of data from another in communicating with the system. To access the information in a file, you need only give its name to UNIX, and the system takes care of locating it. Within certain limitations, you choose for a UNIX file whatever label you care to give it, ideally one which is descriptive of its contents. Names may be up to 14 characters long, but cannot contain imbedded blanks. If you wish to use two words without having to run them together, separate them with another character, as in *chapter.one*. A period is generally a good choice to use for separating characters. You'll probably aim for a balance where filenames are long enough to be descriptive, but short enough to be typed easily and correctly. Generally, you can avoid potential trouble by creating filenames with letters, numbers, and periods (".") only. Many of the other characters have special meanings for UNIX which might confuse the interpretation of a name.

Within these limitations, there's freedom to name files as you wish. Since filenames are used to distinguish one file from another, no two files can have exactly the same name.

Listing the names of files

There may already be a few files associated with your account — the mail that was saved at the end of the last session and probably some files that you've created while practicing with the editor. The first thing we'll do in this session is to find out the names of those files. Login with UNIX, and when the shell is ready to receive instructions, type the list command,

```
% ls
```

Let's say your files are *text*, *notes* and *mbox*, which has the mail you saved from the last session. The response to the ls command would then look like

```
% ls
mbox   notes   text
%
```

Reading a file

There are a few ways to read again the mail you sent to yourself. If it was mailed and saved successfully, one of the files named when you typed ls was *mbox*.

One way to read the file is to use the editor's printing commands:

```
% edit mbox
"mbox" 3 lines, 82 characters
:1,$p
```

Then to save your mail in a file called *letter* you could tell the editor

```
:w letter
"letter" [New file] 3 lines, 82 characters
:q
%
```

Note that ls shows that both *mbox* and *letter* now exist. There is no need to enter the editor to read a file as long as you do not want to use it to make any changes. The command **more** asks the shell to display a file. (The command **1,$p**, to print all of the lines in a file, is only used when you are working in the editor.) To request that UNIX display the mail you sent to yourself, type

```
% more mbox
```

The form of the command is **more** followed by the names of one or more files. **more** displays a screenful of text at a time on CRTs. If more than one filename is given, all of the files are linked together and displayed a screenful at a time. Experiment with

```
% more mbox text
```

(Substitute the name of one of your other files if you have none named *text*)

Copying files

There is a way to save the mail in some other file, say *message*, without using the editor. This can be done by issuing to the shell the command **cp** (for "copy"), which copies one file onto another. The form of the command is **cp file1 file2** to copy *file1* onto *file2*. If the second file already exists, anything that might already be in it is destroyed before the contents of the first file are copied. Otherwise, a new file is created with the second name. Type the commands

```
% cp mbox message
% ls
```

Note that the response from UNIX shows that both *mbox* and *message* now exist.

```
% ls
letter  mbox  message  notes  text
%
```

To verify that the files *message* and *mbox* contain the same text, you can type

```
% cmp message mbox
```

If there are any differences in the two files, **cmp** will print them. Since the files are identical, **cmp** will print nothing.

By the way, **cp** can't be used to copy a file onto itself. Try it to see what happens.

```
% cp message message
```

Removing files

There's no longer any need for the file *mbox* now that there's a copy of it in *message*. Removing files is done with the command **rm**. Type

```
% rm mbox
% ls
```

The output from ls should look something like:

```
% ls
letter  message  notes  text
%
```

Now *mbox* is gone from the list.

Moving files

The same result accomplished by the two commands

```
% cp mbox message
% rm mbox
```

could have been produced, even more simply, by

```
% mv mbox message
```

The command **mv name1 name2** changes the name of the file *name1* to *name2*. If the second file named in the **mv** command already exists, it is removed before the first file is given its name — just as with the **cp** command.

To preserve and protect

To remove the contents of *message* type this instruction to UNIX:

```
% rm message
```

But other commands could also destroy your files accidently. Suppose you already have two different files, *message* and *text*. When UNIX is asked to write new text onto one of your files, it first removes any text already stored there. For example, in response to the request

```
% cp text message
```

UNIX will substitute a copy of *text* for the previous contents of *message*. The command

```
% mv text message
```

destroys whatever was originally in *message* and gives its name to the former contents of the file *text*. You should use the ls command to see the names you have already used before you select a new name for a file.

This is the end of the third session with UNIX. The fourth session will continue with more about UNIX files and their organization.

Session 4: Directories

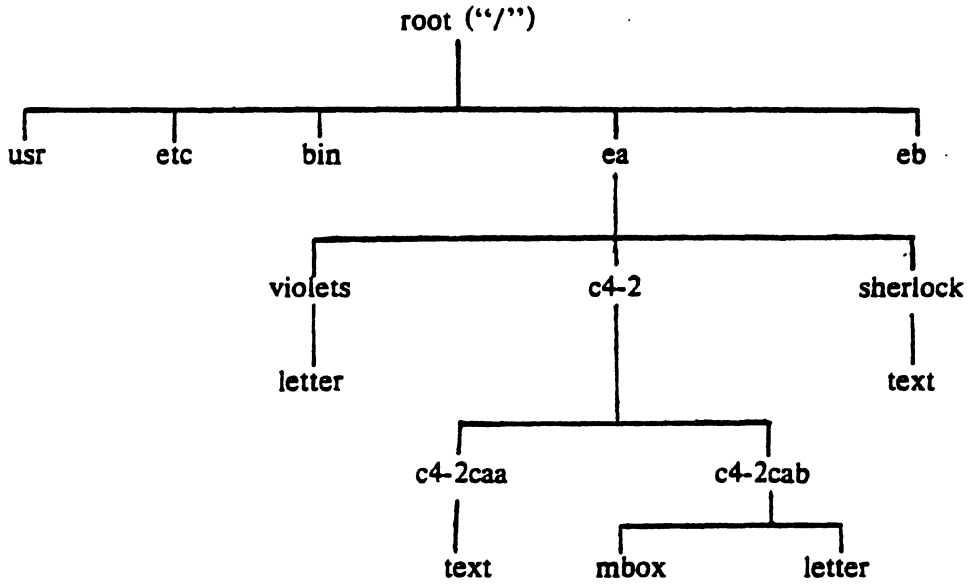
The UNIX file structure

UNIX associates each file with the account of the person who created it as its "owner". This allows different accounts to have files with the same name, because UNIX uses two pieces of information to locate a file: the name of its owner's account and the specific file name.

Files are organized into groups called directories. Directories are valuable because they allow us to keep together such groups of related files as the files belonging to an individual account, the accounts for all the students in a class, or the programs used for a specific project. Each account has its own directory, called its *home* or *login* directory. The files which are created by a user are located within that individual's home directory. Once you login to UNIX, you begin to work "in" your directory; that is, you use the files stored together for your account. A filename is assumed by the system to refer to a file in the same directory as the one you are working in. Thus, if you use the filename *mbox* in a command, UNIX associates the filename with the directory name to locate your file *mbox*.

Directories may contain other directories as well as files of text. Within your home directory, you may wish to create more specialized directories, and each might contain files relating to a separate project.

All files stored on the system are arranged into a hierarchy of directories. UNIX directories are organized in the form of an upside down tree. For each directory or file there is exactly one other directory on the level above it, its "parent," which contains that directory or file. The parent directory may in turn belong to another directory. Eventually, all branches in the tree trace back to one source, the "root" of the tree. The root is symbolized by "/". The UNIX file structure can be represented by a diagram such as this:



The root contains a number of major directories, such as *etc* and *usr*, which contain the programs that make up the UNIX system. Also located within root are the directories of users' files. These directories may have names similar to *ea* or *eb*, depending upon which UNIX system you use (here, we are using the E system).

Pathnames

The command

% pwd

for "print working directory", gives you the full name of the directory you are working in. The system's response to this command will look something like */ea/sherlock* or, perhaps

/ea/c4-2/c4-2caa for a class account. The full name of a directory or file is called its "pathname", since it traces the full "path" from root to the file.

We can trace the path from root to the directory for the account *sherlock*, which has the pathname */ea/sherlock*. The */ea* says that, starting from root, *sherlock*'s account is in the group of user accounts called *ea*. Different levels of the file structure are separated by a slash ("/"). On the next level is the name of the account's directory, *sherlock*. The pathnames for files of text are constructed in the same manner. For example, *sherlock*'s file *text* has the pathname

/ea/sherlock/text

and the one belonging to *c4-2caa* has the pathname

/ea/c4-2/c4-2caa/text

Pathnames can be used wherever files names can be used. Find the pathname for your account with the command **pwd**, and then use it to read one of your files. Use the entire pathname given in response to **pwd**, followed by a "/" and then the name of the file you want to read. For instance if you were *sherlock* and wanted to read your file *text*:

```
% pwd
/ea/sherlock
% more /ea/sherlock/text
```

Assuming that your working directory is */ea/sherlock*, the last command is the equivalent of

```
% more text
```

If your account is *c4-2caa*, you could read your own file named *text*:

```
% pwd
/ea/c4-2/c4-2caa
% more /ea/c4-2/c4-2caa/text
```

Creating a directory

A group of files belonging to a user may be organized into a directory. When you are writing a lengthy paper, each section might be a separate file in the directory *paper*. If, instead, you are writing programs, you may wish to keep a program and a file of input for it together in a directory.

To create the directory *paper*, give the command

```
% mkdir paper
```

(Any valid filename may be used.)

Changing directories

When you want to work in the directory *paper*, use the command **cd** ("change working directory"), followed by the directory's name. If you haven't yet created a directory, you might do so now. Change to the new directory and have its pathname printed:

```
% mkdir paper
% cd paper
% pwd
/ea/sherlock/paper
```

You can always get back to your home directory by using the command

```
% cd
```

without a directory name.

As long as you remain in the directory *paper*, any file name you give will be interpreted as the name of a file in that directory. After you have changed to the new directory, try to read a file in your home directory, such as *text*.

```
% more text
text: No such file or directory
```

UNIX will interpret the command **more text** as a request for the file *text* in *paper*, your current directory. Assuming that you have not yet created any files within *paper*, the system will not find the file specified by your command.

To read *text*, it is necessary to tell UNIX that the file is located in the parent of the directory you are working in. Two dots ("..") are used to symbolize the parent of whichever is the current directory. So the command

```
% more ../text
```

should cause the intended file to be printed.

Similarly, you can change back to the parent of your current directory by typing

```
% cd ..
```

Removing directories

The command to remove a directory is **rmdir** followed by the name of the directory you wish to remove. A directory must be empty before **rmdir** will work. If any files remain in the directory, you will instead be reminded of their existence:

```
% rmdir paper
rmdir: paper not empty
```

Should you forget that *paper* is a directory and try to use the command for ordinary files

```
% rm paper
```

UNIX will print a different message rather than remove the directory. Create a new directory and try it.

If you forget to change back to the parent directory before attempting to remove *paper*, you might get a response like

```
% pwd
/ea/sherlock/paper
% rmdir paper
rmdir: paper non-existent
```

The message reminds you that UNIX was looking for a directory named *paper* within your working directory, *paper*. That is, it tried to find a directory with a pathname like */ea/sherlock/paper/paper* and was unsuccessful.

More about pathnames

Suppose you are working in the directory */ea/sherlock*. To refer to the file named *part3* in your directory *paper*, you could type either

```
% more /ea/sherlock/paper/part3
```

or

```
% more paper/part3
```

You need to specify only that part of the pathname that follows the name of your current directory. Note that the name of a file within your directory starts with the first letter of its name, without the "/" separating the current directory from the filename in the full pathname.

If you want to issue a command concerning a file that is not in your current directory, do so by giving the file's pathname. The ls command is a request to print the names of files within the current directory, but a different directory can be specified by typing its name after the command. For instance, to print the names of the files in the parent of your current directory, type

```
% ls ..
```

The directory specified need not be one of your own. When you type

```
% ls /
```

UNIX will respond with a list of the files and subdirectories stored in the root directory. You may ask to see a file which belongs to another user by specifying its name in a command such as ls or more. When users have indicated that they do not want other people to have access to their files, UNIX will inform you of this rather than execute your command.

This is the end of the fourth session with UNIX.

Session 5: More Commands

The shell's role is to interpret commands and coordinate their execution, and it is able to carry out its duties with flexibility. This session is an introduction to some of the shell's capabilities that can make your work with UNIX easier.

Issuing commands

In the previous sessions when you wanted to give a command, you typed its name on a new line of input to the shell. Should it be more convenient, two or more commands can be typed on a single line if they are separated by a semi-colon (";"). Login, and when the shell is ready to receive instructions type

```
% date; who
```

Your commands will be executed in sequence. After the date and time are printed as requested when you typed **date**, the response to the command **who** will appear as a list of all the users who are currently logged in.

Type-ahead

It is not necessary to wait for a new percent-sign prompt before starting to type a new command. You can begin your next instruction while the last command is still being executed. Although whatever you type will be mixed in with any output that might be sent to your terminal, UNIX will read ahead and store your command correctly. Don't forget to press RETURN. Then, as soon as the system is ready to perform a new task, your command will be interpreted and executed. Issue the command **who** again, and while the response is still being printed, give a second command, perhaps **ls**, to experience this feature of UNIX.

Saving output on a file

The output from most UNIX programs normally appears on the terminal. Let's say, however, that we wanted to create a file named *people* that will contain a list of all the users who are now logged in. The **who** command provides the necessary information, and it is possible to *redirect* output to a file instead of sending it to a terminal. Type the command:

```
% who > people
```

(or substitute some other name if you already have a file called *people*).

The symbol ">" followed by the name of a file tells the system to write the output to that file rather than on the terminal. If you don't yet have a file with the given name, it will be created. If the file already exists, its contents will be replaced by the output produced in response to your command.

Now suppose that we wanted to add the current date and time (that is, the output from the command **date**) to the end of the file we just created. The symbol ">>" followed by a filename is used after the name of a command to indicate that the output is to be appended to the end of that file. The file named after ">>" will be created if it does not already exist. To illustrate this, you can type

```
% date >> people
```

and use *more* to list the file *people* to read the result:

```
% more people
```

Reading input from a file

Many commands, for example **mail**, read information a person normally types at the terminal. It is possible to have a command read this information from a file instead of the terminal. The symbol "<" is typed just before the name of the file which is to provide input.

To see how this works, you can send a short file to yourself as mail.

The file *people*, created in the previous example, can be used as input for the **mail** command. Send the contents of the file to your mailbox and then retrieve it by typing these commands:

```
% mail < people
% mail
```

The line printer

Printed output may be obtained from the high speed line printer as well as from a printing terminal. The command **lpr** is used to request line printer output. When you want to print the contents of one of your files, type its name after the command, as in:

```
% lpr people
Output will be found in Room B6 Evans Hall (box 62).
%
```

The system responds with a message indicating where you can find your output.

Connecting commands with pipe (|)

Commands may be connected so that the output from one program is used as input to the next. The symbol " | " (the vertical bar, which is the shift-\ on the keyboard) placed between two commands tells the shell to use the output from the first program as input to the second. The first program's output is neither printed at the terminal nor recorded in a permanent file — it is used only as input for the execution of the command whose name appears after the " | ".

To illustrate how commands are connected, we can use the command **diff** that shows differences between two files. The form of the command is

```
% diff file1 file2
```

where the name of the chosen files are substituted for *file1* and *file2*. The output of **diff** looks something like a series of editor commands that describe how *file1* would have to be changed to make it the same as *file2*. **Diff** lists the lines that would have to be added, deleted, or changed. Lines reproduced from the first file are marked with "<", and lines from the second are shown with a ">".

To get a feeling for **diff**, make a copy of your file *people*:

```
% cp people newpeople
```

(assuming you don't already have a file called *newpeople*). Next, use the editor to delete one line from *newpeople* and to make a substitution in another line. Quit the editor and type the command

```
% diff people newpeople
```

and you'll see which lines differ between the two files.

If you connect the commands **diff** and **lpr**:

```
% diff people newpeople | lpr
```

the output from the **diff** command will not appear on your terminal, but will be given directly to the **lpr** command as input. As a result, the output from the **diff** command will then be printed for you on the line printer. The above example is the equivalent of this series of commands:

```
% diff people newpeople > differences
% lpr differences
% rm differences
```

The name "pipe" is used for a sequence of commands chained together with a " | ".

The background

Some commands take a while to execute, so you might not always want to wait for one task to be completed before starting the next. Instead, you can have UNIX execute two or more tasks concurrently. Typing an ampersand ("&") at the end of a command tells the system to return control to the terminal immediately after receiving the command. Your command will be run "in the background", leaving you free to issue further instructions from the terminal.

It's generally better to redirect to a file any output from a command that is executed in the background. This prevents the output from being sent to the terminal while you are trying to work on something else.

Give the system a command with a "&" and note its response:

```
% ls > myfiles &
13751
%
```

The identification number 13751 is the process number. UNIX assigns a unique process number to each command that the system receives. Should you want to stop the execution of a command before it is finished, you can type

% kill *number*

giving its process number for *number*. If you type the number correctly, yet the system replies to the **kill** command with the message "No such process", it is likely that the command has already been completed or that you have mistyped the process number.

Characters with special meanings

Let's say that you've stored sections of a paper in individual files with names such as *part1*, *part2*, and *part3*. You now want to have the whole paper printed on the line printer. One way to accomplish this would be to type

```
% lpr part1 part2 part3 &
```

You could produce the same results with

```
% lpr part* &
```

When the shell reads the asterisk ("*") it treats it as a substitute for any string of characters that it might find in a file name. The command above prints all files having names starting with *part* and ending with any sequence of characters. The question mark ("?") can be used in a similar fashion to substitute for any single character.

Although you may not use "*" or "?" very often, it is good to be aware of their special meanings in order to avoid situations where the shell's interpretation of your command is different from what you intended. "*" and "?" should be used with caution. The command

```
rm *
```

will remove all the files from your current directory, which is probably not what you want.

The "magic" characters shown below have special meanings for the UNIX command processor; they should not be used in filenames. In addition, it is best to avoid the use of "−" as the first character of a filename.

* \ ? ^ & / [< > | ' "

Either single (') or double (") quotes enclosing a special character indicates that it is to lose its special significance. You could also precede a special character with a backslash ("\") to have it treated as an ordinary character. These conventions for the interpretation of special characters are applicable when you are working in the shell. The editor has its own set of rules for interpreting special characters.

This is the end of the fifth session with UNIX.

Index

UNIX Command Summary

Computing Services
University of California
Library, 218 Evans Hall
Berkeley, California 94720
415-642-5205

UNX 1.4.2 October 1980

• Login

:login: type your login name on the same line, followed by a carriage return.

Select Host: bussiplexer only. Type **a** for UNIX A, **b** for UNIX B, etc., followed by a carriage retu

Password: printing is turned off as you enter your password on the same line, followed by a carriage return.

• Logout

logout logs you off the system. You can also logout by typing a control-d: depress the contr key (CTRL) and type **d** simultaneously.

• General commands

mail retrieves mail which has been sent to you, and prints one message at a time, prompt with "?" for disposition. Typing **?** at this point prints a list of mail commands.

mail *name* sends mail to another user. Here, *name* is the login name of the person to receive mail. Then type your message starting on a new line, and end it with a control-d alo on a line; depress control (CTRL) and type **d** simultaneously.

passwd *login-name* changes password. You are prompted once for your current password and twice for y new one. Printing is turned off while the passwords are typed. New passwords may b as long as eight characters, and can include any characters but "#" and "@".

who lists users who are currently logged in. Typing **who am i** limits the report to login information for your terminal only.

• Online documentation

man *command* prints a writeup from the *UNIX Programmer's Manual* for the command specified.

help *topic* provides information about the system. Type **help index** for a list of available topics and then **help** followed by the name of a listed topic for information on that subject. **help news** prints the latest system news.

• Accounting

pq reports on disk storage quota (print quota): (number of blocks currently used/maximum number of blocks allotted to the account). A block is 512 bytes, or characters.

jobno gives the account's job number (account number), which is used to file lineprinter output and for accounting.

Type the commands that are printed here in **boldface** exactly as shown, and supply additional information, if any, described here in *italics*.

• File manipulation

ls	prints a list of the files in the current directory.
ls *directory*	prints a list of the files in the specified directory.
cat *filename(s)*	prints the contents of the named file(s) on the terminal.
lpr *filename(s)*	prints the contents of the named file(s) on the system lineprinter. Type **help printer** f details on where to find output.
cp *file1 file2*	copies *file1* onto *file2*. There will be two separate copies of the file. See warning below
cp *file(s) directory*	makes a copy of the named file(s) in the given directory and gives it the same filename(s) as the original file(s). See warning below.
rm *filename(s)*	removes the named file(s).
mv *oldname newname*	changes the name of a file from *oldname* to *newname*. See warning below.
mv *file(s) directory*	moves file(s) to the specified directory. The original filename(s) is retained. See warning below.

Warning: When using **cp** and **mv**, ensure that a file with a name the same as the "target" filename does not exist already. If it does, its contents will be destroyed before the command is executed.

• Directories

pwd	gives the pathname of the current directory (print working directory).
mkdir *name*	makes a new directory.
chdir *directory-name* or **cd** *directory-name*	changes to a different working directory.
rmdir *directory-name*	removes the indicated directory. The directory must be empty.

• Special characters

The characters shown below have special meanings for the UNIX command processor; they should not be used in filenames. For further information about how these characters are interpreted, see the *UNIX Programmer's Manual*, under "csh(1)." In addition, it is best to avoid using "-" as the first character of a filename.

* \ ? ~ & / [< > | ' "

To interrupt and terminate execution of a command, press RUBOUT or DELETE, sometimes labeled RUB or DEL. On an ADM terminal, hold down SHIFT while typing RUB. Type control-s to stall printing at the terminal, and control-q subsequently to restart printing.

Type the commands that are printed here in **boldface** exactly as shown, and supply additional information, if any, described here in *italics*.

Edit: A Tutorial

Ricki Blau

James Joyce

Computing Services
University of California
Berkeley, California 94720

ABSTRACT

This narrative introduction to the use of the text editor *edit* assumes no prior familiarity with computers or with text editing. Its aim is to lead the beginning UNIX† user through the fundamental steps of writing and revising a file of text. Edit, a version of the text editor *ex*, was designed to provide an informative environment for new and casual users.

We welcome comments and suggestions about this tutorial and the UNIX documentation in general.

September 1981

†UNIX is a trademark of Bell Laboratories.

Contents

Introduction

Text editing using a terminal connected to a computer allows you to create, modify, and print text easily. A *text editor* is a program that assists you as you create and modify text. The text editor you will learn here is named *edit*. Creating text using edit is as easy as typing it on an electric typewriter. Modifying text involves telling the text editor what you want to add, change, or delete. You can review your text by typing a command to print the file contents as they were entered by you. Another program, a text formatter, rearranges your text for you into "finished form." This document does not discuss the use of a text formatter.

These lessons assume no prior familiarity with computers or with text editing. They consist of a series of text editing sessions which lead you through the fundamental steps of creating and revising text. After scanning each lesson and before beginning the next, you should practice the examples at a terminal to get a feeling for the actual process of text editing. If you set aside some time for experimentation, you will soon become familiar with using the computer to write and modify text. In addition to the actual use of the text editor, other features of UNIX will be very important to your work. You can begin to learn about these other features by reading "Communicating with UNIX" or one of the other tutorials that provide a general introduction to the system. You will be ready to proceed with this lesson as soon as you are familiar with (1) your terminal and its special keys, (2) the login procedure, (3) and the ways of correcting typing errors. Let's first define some terms:

program — A set of instructions, given to the computer, describing the sequence of steps the computer performs in order to accomplish a specific task. The tasks must be specific, such as balancing your checkbook or editing your text. A general task, such as working for world peace, is something we can do, but not something we can write programs to do.

UNIX — UNIX is a special type of program, called an operating system, that supervises the machinery and all other programs comprising the total computer system.

edit — *edit* is the name of the UNIX text editor you will be learning to use, and is a program that aids you in writing or revising text. Edit was designed for beginning users, and is a simplified version of an editor named *ex*.

file — Each UNIX account is allotted space for the permanent storage of information, such as programs, data or text. A file is a logical unit of data, for example, an essay, a program, or a chapter from a book, which is stored on a computer system. Once you create a file, it is kept until you instruct the system to remove it. You may create a file during one UNIX session, end the session, and return to use it at a later time. Files contain anything you choose to write and store in them. The sizes of files vary to suit your needs; one file might hold only a single number, yet another might contain a very long document or program. The only way to save information from one session to the next is to store it in a file, which you will learn in Session 1.

filename — Filenames are used to distinguish one file from another, serving the same purpose as the labels of manila folders in a file cabinet. In order to write or access information in a file, you use the name of that file in a UNIX command, and the system will automatically locate the file.

disk — Files are stored on an input/output device called a disk, which looks something like a stack of phonograph records. Each surface is coated with a material similar to the coating on magnetic recording tape, and information is recorded on it.

buffer — A temporary work space, made available to the user for the duration of a session of text editing and used for creating and modifying the text file. We can think of the buffer as a blackboard that is erased after each class, where each session with the editor is a class.

Session 1

Making contact with UNIX

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 for the four ways you can make contact: on a terminal connected to the port selector, on a terminal connected to the bussiplexer, on a terminal that is directly linked to the computer, or over a telephone line where the computer answers your call.

Port selector terminals

If your terminal has a small box attached with a red button on top, it is connected to the port selector. You can request any of several UNIX systems by turning on the terminal and pressing the red button once. The terminal will respond with

Request:

Type the name of the system you are to use. For this example, we use Computer Facilities and Operations (CF+O) System E:

Request:e *(and press RETURN)*

The terminal will skip a line, emit a beep, and then print a greeting inviting you to login. You are now ready to login.

Bussiplexer terminals

The bussiplexer is a communications network that makes it possible for you to connect to any of several CF&O UNIX systems. Turn on the terminal. If you don't immediately receive a login message, press RETURN. The login message will be similar to:

U.C. Bussiplexer (A+B+C+D+E+F+)

:login:

Type your login name and press RETURN. For most accounts, the bussiplexer will respond:

Connecting to Unix X

where X is the UNIX system that your account is on. If the bussiplexer does not know which of the systems has your account, you will be asked to select one. Type a single letter for the system of your choice (for instance, "e" if your account is on UNIX E) followed by a RETURN, as in this example:

Select Host: e< *cr*>
Connecting to Unix E

Connected

After a few seconds, your UNIX system will ask for a password:

Password: *(type your password and press RETURN)*

You are now logged in.

Directly-linked terminals

Turn on your terminal and press the RETURN key. You are now ready to login.

Dial-up terminals

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. You are now ready to login.

Logging in

The message inviting you to login is:

:login:

Type your login name, which identifies you to UNIX, on the same line as the login message, and press 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 the RETURN key)*

(In the examples, input you would type 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 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 you are about to create. To begin your editing session, type **edit** followed by a space and then the filename you have selected; for example, "text". When you have completed the command, press the RETURN key and wait for edit's response:

% edit text *(followed by a RETURN)*
"text" No such file or directory
:

If you typed the command correctly, you will now be in communication with edit. Edit has set aside a buffer for use as a temporary working space during your current editing session. It also checked to see if the file you named, "text", already existed. It was unable to find such a file, since "text" is a new file we are about to create. Edit confirms this with the line:

"text" No such file or directory

On the next line appears edit's prompt ":", announcing that you are in *command mode* and edit

expects a command from you. You may now begin to create the new file.

The "Command not found" message

If you misspelled edit by typing, say, "editor", your request would be handled as follows:

```
% editor
editor: Command not found
%
```

Your mistake in calling edit "editor" was treated by UNIX as a request for a program named "editor". Since there is no program named "editor", UNIX reported that the program was "not found". A new % indicates that UNIX is ready for another command, and you may then enter the correct command.

A summary

Your exchange with UNIX as you logged in and made contact with edit should look something like this:

```
:login: susan
Password:
... A Message of General Interest ...
% edit text
"text" No such file or directory
:
```

Entering text

You may now begin entering text into the buffer. This is done by *appending* (or adding) text to whatever is currently in the buffer. Since there is nothing in the buffer at the moment, you are appending text to nothing; in effect, since you are adding text to nothing you are creating text. Most edit commands have two forms: a word that suggests 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 at first, but once you are familiar with editing you may prefer to type the shorter abbreviations. The command to input text is "append", and it may be abbreviated "a". Type **append** and press the RETURN key.

```
% edit text
: append
```

Messages from *edit*

If you make a mistake in entering a command and type something that edit does not recognize, edit will respond with a message intended to help you diagnose your error. For example, if you misspell the command to input text by typing, perhaps, "add" instead of "append" or "a", you will receive this message:

```
: 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 execute 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 stops sending you a prompt. You will not receive any prompts or error messages while in text input mode. 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 the RETURN key.* When you type the period and press RETURN, you signal that you want to stop appending text, and edit responds by allowing you to exit 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 the RETURN key.

This is a good place 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 RETURN that gets you out of append mode.

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 as many times as there are characters you want to erase.

The usual erase character is the backspace (control-H), and you can correct typing errors in the line you are typing by holding down the CTRL key and typing the "H" key. If you try typing control-H you will notice that the terminal backspaces in the line you are on. You can backspace over your error, and then type what you want to be the rest of the line.

If you make a bad start in a line and would like to begin again, you can either backspace to the beginning of the line or you can use the at-sign "@" to erase everything on the line:

```
Text edtiing is strange, but@
Text editing is strange, but nice.
```

When you type the at-sign (@), you erase the entire line typed so far and are given a fresh line to type on. You may immediately begin to retype the line. (If you are on a bussiplexer terminal, you will not advance to the next line after typing "@", but there is no other difference in the way the at-sign is handled.) This, unfortunately, does not help after you type the line and press RETURN. To make corrections in lines that have been completed, it is necessary to use the editing commands covered in the next session and those that follow.

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. Learning how to write a file to disk is second in importance only to entering the text. To write the contents of the buffer to a disk file, use the command "write" (or its abbreviation "w"):

```
:write
```

Edit will copy the contents of the buffer to a disk file. If the file does not yet exist, a new file will be created automatically and the presence of a "[New file]" will be noted. The newly-created file will be given the name specified when you entered the editor, in this case "text". To confirm that the disk file has been successfully written, edit will repeat the filename and give the number of lines and the total number of characters in the file. The buffer remains unchanged by the "write" command. All of the lines that were written to disk will still be in the buffer, should you want to modify or add to them.

Edit must have a 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 RETURN:

```
:write
"text" [New file] 3 lines, 90 characters
:quit
%
```

The % is from UNIX to tell you that your session with edit is over and you may command UNIX further. Since we want to end the entire session at the terminal, we also need to exit from UNIX. In response to the UNIX prompt of "%" type the command

```
% logout
```

This will end your session with UNIX, and will ready the terminal for the next user. It is always important to type **logout** at the end of a session to make absolutely sure no one could accidentally stumble into your abandoned session and thus gain access to your files, tempting even the most honest of souls.

This is the end of the first session on UNIX text editing.

Session 2

Login with UNIX as in the first session:

```
:login: susan  (carriage return)
Password:      (give password and carriage return)
... A Message of General Interest ...
%
```

When you indicate 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, and indicates this by its prompt character, the colon (:). 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. When "append" is the first command of your editing session, the lines you enter are placed at the end of the buffer. 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.
.
```

You may recall that once you enter append mode using the "a" (or "append") command, you need to type a line containing only a period (.) to exit append mode.

Interrupt

Should you press the RUB 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 rub 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 you were typing when the append command was interrupted will not be entered into the buffer.

Making corrections

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 either to the last few pages of Session 1 or to "Communicating with UNIX" if you need to review the procedures for making a correction. The most important idea to remember is that erasing a character or cancelling a line must be done before you press the RETURN key.

Listing what's in the buffer (p)

Having appended text to what you wrote in Session 1, you might want to see all the lines in the buffer. To print the contents of the buffer, type the command:

:1,$p

The "1"† stands for line 1 of the buffer, the "$" is a special symbol designating the last line of the buffer, and "p" (or **print**) is the command to print from line 1 to the end of the buffer. The command "1,$p" gives you:

```
This is some sample text.
And thiss is some more text.
Text editing is strange, but nice.
This is text added in Session 2.
It doesn't mean much here, but
it does illustrate the editor.
```

Occasionally, you may accidentally type a character that can't be printed, which can be done by striking a key while the CTRL key is pressed. In printing lines, edit uses a special notation to show the existence of non-printing characters. Suppose you had introduced the non-printing character "control-A" into the word "illustrate" by accidently pressing the CTRL key while typing "a". This can happen on many terminals because the CTRL key and the "A" key are beside each other. If your finger presses between the two keys, control-A results. When asked to print the contents of the buffer, edit would display

```
it does illustr^Ate the editor.
```

To represent the control-A, edit shows "^A". The sequence "^" followed by a capital letter stands for the one character entered by holding down the CTRL key and typing the letter which appears after the "^". We'll soon discuss the commands that can be used to correct this typing error.

In looking over the text we see that "this" is typed as "thiss" in the second line, a deliberate error so we can learn to make corrections. Let's correct the spelling.

Finding things in the buffer

In order to change something in the buffer we first need to find it. We can find "thiss" in the text we have entered by looking at a listing of the lines. Physically speaking, we search the lines of text looking for "thiss" and stop searching when we have found it. The way to tell edit to search for something is to type it inside slash marks:

:/thiss/

By typing **/thiss/** and pressing RETURN, you instruct edit to search for "thiss". If you ask edit to look for a pattern of characters which it cannot find in the buffer, it will respond "Pattern not found". When edit finds the characters "thiss", it will print the line of text for your inspection:

```
And thiss is some more text.
```

Edit is now positioned in the buffer at the line it just printed, ready to make a change in the line.

---

†The numeral "one" is the top left-most key, and should not be confused with the letter "el".

The current line

Edit keeps track of the line in the buffer where it is located at all times during an editing session. In general, the line that has been most recently printed, entered, or changed is the current location in the buffer. The editor is prepared to make changes at the current location in the buffer, unless you direct it to another location.

In particular, when you bring a file into the buffer, you will be located at the last line in the file, where the editor left off copying the lines from the file to the buffer. If your first editing command is "append", the lines you enter are added to the end of the file, after the current line — the last line in the file.

You can refer to your current location 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 press RETURN, and edit will respond with the line number:

```
:.=
2
```

If you type the number of any line and press 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 gain experience in using them to make changes.

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.
```

Note 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 their corresponding line numbers.

Substitute command (s)

Now that you have found the misspelled word, you can 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 edit to make a 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 it 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 that your instructions could not be carried out. When edit does find the characters that you want to change, it will make the substitution and automatically print the changed line, so that you can check that the correct substitution was made. In the example,

:2s/thiss/this/
And this is some more text.

line 2 (and line 2 only) will be searched for the characters "thiss", and when the first exact match is found, "thiss" will be changed to "this". Strictly speaking, it was not necessary above to specify the number of the line to be changed. In

:s/thiss/this/

edit will assume that we mean to change the line where we are currently located ("."). In this case, the command without a line number would have produced the same result because we were already located at the line we wished to change.

For another illustration of the substitute command, let us choose the line:

Text editing is strange, but nice.

You can make this line a bit more positive by taking out the characters "strange, but " so the line reads:

Text editing is nice.

A command that will first position edit at the desired line and then make the substitution is:

:/strange/s/strange, but //

What we have done here is combine our search with our substitution. Such combinations are perfectly legal, and speed up editing quite a bit once you get used to them. That is, you do not necessarily have to use line numbers to identify a line to edit. Instead, you may identify the line you want to change by asking edit to search for a specified pattern of letters that occurs in that line. The parts of the above command are:

/strange/	tells edit to find the characters "strange" in the text
s	tells edit 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 that the space is to be taken out, your line will read:

Text editing is nice.

which looks a little funny because of the extra space between "is" and "nice". Again, we realize from this that a blank space is a real character to a computer, and in editing text we need to be aware of spaces within a line just as we would be aware of an "a" or a "4".

Another way to list what's in the buffer (z)

Although the print command is useful for looking at specific lines in the buffer, other commands may be more convenient for viewing large sections of text. You can ask to see a screen full of text at a time by using the command **z**. If you type

:1z

edit will start with line 1 and continue printing lines, stopping either when the screen of your terminal is full or when the last line in the buffer has been printed. If you want to read the

next segment of text, type the command

```
: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 (:quit! overrides)
:
```

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 did during the editing session since you typed the latest write command. Because in this lesson we have not written to disk at all, everything we have done would have been lost if edit had obeyed the **q** command. If you did not want to save the work done during this editing session, you would have to type "q!" or ("quit!") to confirm that you indeed wanted to end the session immediately, leaving the file as it was after the most recent "write" command. However, since you want to save what you have edited, you need to type:

```
:w
"text" 6 lines, 171 characters
```

and then follow with the commands to quit and logout:

```
:q
% logout
```

and hang up the phone or turn off the terminal when UNIX asks for a name. Terminals connected to the port selector will stop after the logout command, and pressing keys on the keyboard will do nothing.

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 type

```
% edit text
```

or simply

```
% edit
```

Both ways get you in contact with edit, but the first way will bring a copy of the file named "text" into the buffer. If you did forget to tell edit the name of your file, you can get it into the buffer by typing:

```
:e text
"text" 6 lines, 171 characters
```

The command **edit**, which may be abbreviated **e**, tells edit that you want to erase anything that might already be in the buffer and bring a copy of the file "text" into the buffer for editing. You may also use the edit (e) command to change files in the middle of an editing session, or to give edit the name of a new file that you want to create. Because the edit command clears the buffer, you will receive a warning if you try to edit a new file without having saved a copy of the old file. This gives you a chance to write the contents of the buffer to disk before editing the next file.

Moving text in the buffer (m)

Edit allows you to move lines of text from one location in the buffer to another by means of the **move (m)** command. The first two examples are for illustration only, though after you have read this Session you are welcome to return to them for practice. The command

```
:2,4m$
```

directs edit to move lines 2, 3, and 4 to the end of the buffer ($). The format for the move command is that you specify the first line to be moved, the last line to be moved, the move command "m", and the line after which the moved text is to be placed. So,

```
:1,3m6
```

would instruct edit to move lines 1 through 3 (inclusive) to a location after line 6 in the buffer. To move only one line, say, line 4, to a location in the buffer after line 5, the command would be "4m5".

Let's move some text using the command:

```
:5,$m1
2 lines moved
it does illustrate the editor.
```

After executing a command that moves more than one line of the buffer, edit tells how many lines were affected by the move and prints the last moved line for your inspection. If you want to see more than just the last line, you can then use the print (p), z, or number (nu) command to view more text. The buffer should now contain:

This is some sample text.
It doesn't mean much here, but
it does illustrate the editor.
And this is some more text.
Text editing is nice.
This is text added in Session 2.

You 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/

(Do not type both examples here!) The problem with combining context searching with the move command is that your chance of making a typing error in such a long command is greater than if you type 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:

:2,5copy $

makes a copy of lines 2 through 5, 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, which is "This is text added in Session 2." if you typed the commands suggested so far.

: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 that 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 line containing the indicated text, starting its search at the current line and moving line by line until it finds the text. Once you are sure that you have correctly specified the line 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.
And this is some more text.
Text editing is nice.
This is text added in Session 2.
It doesn't mean much here, but
```

To delete both lines 2 and 3:

```
And this is some more text.
Text editing is nice.
```

you type

```
:2,3d
2 lines deleted
```

which specifies the range of lines from 2 to 3, and the operation on those lines — "d" for delete. If you delete more than one line you will receive a message telling you the number of lines deleted, as indicated in the example above.

The previous example assumes that you know the line numbers for the lines to be deleted. If you do not you might combine the search command with the delete command:

```
:/And this is some/,/Text editing is nice./d
```

A word or two of caution

In using the search function to locate lines to be deleted you should be **absolutely sure** the characters you give as the basis for the search will take edit to the line you want deleted. Edit will search for the first occurrence of the characters starting from where you last edited — that is, from the line you see printed if you type dot (.).

A search based on too few characters may result in the wrong lines being deleted, which edit will do as easily as if you had meant it. For this reason, it is usually safer to specify the search and then delete in two separate steps, at least until you become familiar enough with using the editor that you understand how best to specify searches. For a beginner it is not a bad idea to double-check each command before pressing RETURN to send the command on its way.

Undo (u) to the rescue

The **undo (u)** command has the ability to reverse the effects of the last command that changed the buffer. To undo the previous command, type "u" or "undo". Undo can rescue the contents of the buffer from many an unfortunate mistake. However, its powers are not unlimited, so it is still wise to be reasonably careful about the commands you give.

It is possible to undo only commands which have the power to change the buffer — for example, delete, append, move, copy, substitute, and even undo itself. The commands write (w) and edit (e), which interact with disk files, cannot be undone, nor can commands that do not change the buffer, such as print. Most importantly, the **only** command that can be reversed by undo is the last "undo-able" command you typed. You can use control-H and @ to change commands while you are typing them, and undo to reverse the effect of the commands after you have typed them and pressed RETURN.

To illustrate, let's issue an undo command. Recall that the last buffer-changing command we gave deleted the lines formerly numbered 2 and 3. Typing undo at this moment will reverse the effects of the deletion, causing those two lines to be replaced in the buffer.

```
: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 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:

```
:.=
```

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 "$", then, 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 −)

When you are editing you often want to go back and re-read a previous line. You could specify a context search for a line you want to read if you remember some of its text, but if you simply want to see what was written a few, say 3, lines ago, you can type

```
−3p
```

This tells edit to move back to a position 3 lines before the current line (.) and print that line. You can move forward in the buffer similarly:

```
+2p
```

instructs edit to print the line that is 2 ahead of your 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 ($), and the original lines referred to by "−1" and "+2" remain where they are.

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 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
```

or

```
Not that many lines in buffer
```

Similarly, if you try to move to a position before the first line, edit will print one of these messages:

```
Nonzero address required on this command
```

or

```
Negative address – first buffer line is 1
```

The number associated with a buffer line is the line's "address", in that it can be used to locate the line.

Changing lines (c)

You can also delete certain lines and insert new text in their place. This can be accomplished easily with the **change** (**c**) command. The change command instructs edit to delete specified lines and then switch to text input mode to accept the text that will replace them. Let's say you want to change the first two lines in the buffer:

```
This is some sample text.
And this is some more text.
```

to read

```
This text was created with the UNIX text editor.
```

To do so, you type:

```
:1,2c
2 lines changed
This text was created with the UNIX text editor.
.
:
```

In the command **1,2c** we specify that we want to change the range of lines beginning with 1 and ending with 2 by giving line numbers as with the print command. These lines will be deleted. After you type RETURN to end the change command, edit notifies you if more than one line will be changed and places you in text input mode. Any text typed on the following lines will be inserted into the position where lines were deleted by the change command. **You will remain in text input mode until you exit in the usual way, by typing a period alone on a line.** Note that the number of lines added to the buffer need not be the same as the number of lines deleted.

This is the end of the third session on text editing with UNIX.

Session 4

This lesson covers several topics, starting with commands that 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 for 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:

:5s/text/material/g

to change every instance of "text" in line 5 alone. Further, neither command will change "text" to "material" if "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

You should be careful 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 file 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 "text" to "texts" we may type either

```
:/text/s/text/texts/
```

as we have done in the past, or a somewhat abbreviated command:

```
:/text/s//texts/
```

In this example, the characters to be changed are not specified — there are no characters, not even a space, between the two slash marks that indicate what is to be changed. This lack of characters between the slashes is taken by the editor to mean "use the characters we last searched for as the characters to be changed."

Similarly, the last context search may be repeated by typing a pair of slashes with nothing between them:

```
:/does/
It doesn't mean much here, but
://
it does illustrate the editor.
```

(You should note that the search command found the characters "does" in the word "doesn't" in the first search request.) Because no characters are specified for the second search, the editor scans the buffer for the next occurrence of the characters "does".

Edit normally searches forward through the buffer, wrapping around from the end of the buffer to the beginning, until the specified character string is found. If you want to search in the reverse direction, use question marks (?) instead of slashes to surround the characters you are searching for.

It is also possible to repeat the last substitution without having to retype the entire command. An ampersand (&) used as a command repeats the most recent substitute command, using the same search and replacement patterns. After altering the current line by typing

```
:s/text/texts/
```

you type

```
:/text/&
```

or simply

```
://&
```

to make the same change on the next line in the buffer containing the characters "text".

Special characters

Two characters have special meanings when used in specifying searches: "$" and "^". "$" is taken by the editor to mean "end of the line" and is used to identify strings that occur at the end of a line.

```
:g/text.$/s//material./p
```

tells the editor to search for all lines ending in "text." (and nothing else, not even a blank space), to change each final "text." to "material.", and print the changed lines.

The symbol "^" indicates the beginning of a line. Thus,

```
:s/^/1. /
```

instructs the editor to insert "1." and a space at the beginning of the current line.

The characters "$" and "^" have special meanings only in the context of searching. At other times, they are ordinary characters. If you ever need to search for a character that has a special meaning, you must indicate that the character is to lose temporarily its special significance by typing another special character, the backslash (\), before it.

```
: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 shell command. If the buffer contents have not been written since the last change, a warning will be printed before the command is executed:

```
[No write since last change]
```

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. If you are editing a file named "draft3" having 283 lines in it, you can have the editor write onto a different file by including its name in the write command:

```
:w chapter3
"chapter3" [new file] 283 lines, 8698 characters
```

The current filename remembered by the editor *will not be changed as a result of the write command.* Thus, if the next write command does not specify a name, edit will write onto the current file ("draft3") 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, the number of lines in the buffer, and the percent of the distance through the file your current location is.

```
: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 at a specified location, essentially copying new lines between two existing lines. To use it, specify the line after which the new text will be placed, the **read (r)** command, and then the name of the file. If you have a file named "example", the command

```
:$r example
"example" 18 lines, 473 characters
```

reads the file "example" and adds it to the buffer after the last line. The current filename is not changed by the read command.

Writing parts of the buffer

The **write (w)** command can write all or part of the buffer to a file you specify. We are already familiar with writing the entire contents of the buffer to a disk file. To write only part of the buffer onto a file, indicate the beginning and ending lines before the write command, for example

```
:45,$w ending
```

Here all lines from 45 through the end of the buffer are written onto the file named *ending.* The lines remain in the buffer as part of the document you are editing, and you may continue to edit the entire buffer. Your original file is unaffected by your command to write part of the buffer to another file. Edit still remembers whether you have saved changes to the buffer in your original file or not.

Recovering files

Although it does not happen very often, there are times UNIX stops working because of some malfunction. This situation is known as a *crash.* Under most circumstances, edit's crash recovery feature is able to save work to within a few lines of changes before a crash (or an accidental phone hang up). If you lose the contents of an editing buffer in a system crash, you will normally receive mail when you login that gives the name of the recovered file. To recover the file, enter the editor and type the command **recover (rec)**, followed by the name of the lost file. For example, to recover the buffer for an edit session involving the file "chap6", the command is:

```
:recover chap6
```

Recover is sometimes unable to save the entire buffer successfully, so always check the contents of the saved buffer carefully before writing it back onto the original file. For best results, write the buffer to a new file temporarily so you can examine it without risk to the original file. Unfortunately, you cannot use the recover command to retrieve a file you removed using the shell command **rm.**

Other recovery techniques

If something goes wrong when you are using the editor, it may be possible to save your work by using the command **preserve (pre)**, which saves the buffer as if the system had crashed. If you are writing a file and you get the message "Quota exceeded", you have tried to

use more disk storage than is allotted to your account. *Proceed with caution* because it is likely that only a part of the editor's buffer is now present in the file you tried to write. In this case you should use the shell escape from the editor (!) to remove some files you don't need and try to write the file again. If this is not possible and you cannot find someone to help you, enter the command

: **preserve**

and wait for the reply,

File preserved.

If you do not receive this reply, seek help immediately. Do not simply leave the editor. If you do, the buffer will be lost, and you may not be able to save your file. If the reply is "File preserved." you can leave the editor (or logout) to remedy the situation. After a preserve, you can use the recover command once the problem has been corrected, or the **-r** option of the edit command if you leave the editor and want to return.

If you make an undesirable change to the buffer and type 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, but a selection of commands that 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 Computing Services 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 from 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.

Index

A Tutorial Introduction to the UNIX Text Editor

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Almost all text input on the UNIX† operating system is done with the text-editor *ed*. This memorandum is a tutorial guide to help beginners get started with text editing.

Although it does not cover everything, it does discuss enough for most users' day-to-day needs. This includes printing, appending, changing, deleting, moving and inserting entire lines of text; reading and writing files; context searching and line addressing; the substitute command; the global commands; and the use of special characters for advanced editing.

September 21, 1978

†UNIX is a Trademark of Bell Laboratories.

A Tutorial Introduction to the UNIX Text Editor

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

Ed is a "text editor", that is, an interactive program for creating and modifying "text", using directions provided by a user at a terminal. The text is often a document like this one, or a program or perhaps data for a program.

This introduction is meant to simplify learning *ed*. The recommended way to learn *ed* is to read this document, simultaneously using *ed* to follow the examples, then to read the description in section I of the *UNIX Programmer's Manual*, all the while experimenting with *ed*. (Solicitation of advice from experienced users is also useful.)

Do the exercises! They cover material not completely discussed in the actual text. An appendix summarizes the commands.

Disclaimer

This is an introduction and a tutorial. For this reason, no attempt is made to cover more than a part of the facilities that *ed* offers (although this fraction includes the most useful and frequently used parts). When you have mastered the Tutorial, try *Advanced Editing on UNIX*. Also, there is not enough space to explain basic UNIX procedures. We will assume that you know how to log on to UNIX, and that you have at least a vague understanding of what a file is. For more on that, read *UNIX for Beginners*.

You must also know what character to type as the end-of-line on your particular terminal. This character is the RETURN key on most terminals. Throughout, we will refer to this character, whatever it is, as RETURN.

Getting Started

We'll assume that you have logged in to your system and it has just printed the prompt character, usually either a $ or a %. The easiest way to get *ed* is to type

```
ed        (followed by a return)
```

You are now ready to go — *ed* is waiting for you to tell it what to do.

Creating Text — the Append command "a"

As your first problem, suppose you want to create some text starting from scratch. Perhaps you are typing the very first draft of a paper; clearly it will have to start somewhere, and undergo modifications later. This section will show how to get some text in, just to get started. Later we'll talk about how to change it.

When *ed* is first started, it is rather like working with a blank piece of paper — there is no text or information present. This must be supplied by the person using *ed*: it is usually done by typing in the text, or by reading it into *ed* from a file. We will start by typing in some text, and return shortly to how to read files.

First a bit of terminology. In *ed* jargon, the text being worked on is said to be "kept in a buffer." Think of the buffer as a work space, if you like, or simply as the information that you are going to be editing. In effect the buffer is like the piece of paper, on which we will write things, then change some of them, and finally file the whole thing away for another day.

The user tells *ed* what to do to his text by typing instructions called "commands." Most commands consist of a single letter, which must be typed in lower case. Each command is typed on a separate line. (Sometimes the command is preceded by information about what line or lines of text are to be affected — we will discuss these shortly.) *Ed* makes no response to most commands — there is no prompting or typing of messages like "ready". (This silence is preferred by experienced users, but sometimes a hangup for beginners.)

The first command is *append*, written as the letter

```
a
```

all by itself. It means "append (or add) text lines to the buffer, as I type them in." Appending is rather like writing fresh material on a piece of paper.

So to enter lines of text into the buffer, just type an *a* followed by a RETURN, followed by

the lines of text you want, like this:

```
a
Now is the time
for all good men
to come to the aid of their party.
.
```

The only way to stop appending is to type a line that contains only a period. The "." is used to tell *ed* that you have finished appending. (Even experienced users forget that terminating "." sometimes. If *ed* seems to be ignoring you, type an extra line with just "." on it. You may then find you've added some garbage lines to your text, which you'll have to take out later.)

After the append command has been done, the buffer will contain the three lines

```
Now is the time
for all good men
to come to the aid of their party.
```

The "a" and "." aren't there, because they are not text.

To add more text to what you already have, just issue another a command, and continue typing.

Error Messages — "?"

If at any time you make an error in the commands you type to *ed*, it will tell you by typing

```
?
```

This is about as cryptic as it can be, but with practice, you can usually figure out how you goofed.

Writing text out as a file — the Write command "w"

It's likely that you'll want to save your text for later use. To write out the contents of the buffer onto a file, use the *write* command

```
w
```

followed by the filename you want to write on. This will copy the buffer's contents onto the specified file (destroying any previous information on the file). To save the text on a file named **junk**, for example, type

```
w junk
```

Leave a space between w and the file name. *Ed* will respond by printing the number of characters it wrote out. In this case, *ed* would respond with

```
68
```

(Remember that blanks and the return character at the end of each line are included in the character count.) Writing a file just makes a copy of the text — the buffer's contents are not disturbed, so you can go on adding lines to it. This is an important point. *Ed* at all times works on a copy of a file, not the file itself. No change in the contents of a file takes place until you give a w command. (Writing out the text onto a file from time to time as it is being created is a good idea, since if the system crashes or if you make some horrible mistake, you will lose all the text in the buffer but any text that was written onto a file is relatively safe.)

Leaving ed — the Quit command "q"

To terminate a session with *ed*, save the text you're working on by writing it onto a file using the w command, and then type the command

```
q
```

which stands for *quit*. The system will respond with the prompt character ($ or %). At this point your buffer vanishes, with all its text, which is why you want to write it out before quitting.†

Exercise 1:

Enter *ed* and create some text using

```
a
. . . text . . .
.
```

Write it out using w. Then leave *ed* with the q command, and print the file, to see that everything worked. (To print a file, say

```
pr filename
```

or

```
cat filename
```

in response to the prompt character. Try both.)

Reading text from a file — the Edit command "e"

A common way to get text into the buffer is to read it from a file in the file system. This is what you do to edit text that you saved with the w command in a previous session. The *edit* command e fetches the entire contents of a file into the buffer. So if you had saved the three lines "Now is the time", etc., with a w command in an earlier session, the *ed* command

```
e junk
```

would fetch the entire contents of the file **junk** into the buffer, and respond

† Actually, *ed* will print ? if you try to quit without writing. At that point, write if you want; if not, another q will get you out regardless.

```
68
```

which is the number of characters in **junk**. *If anything was already in the buffer, it is deleted first.*

If you use the e command to read a file into the buffer, then you need not use a file name after a subsequent w command; *ed* remembers the last file name used in an e command, and w will write on this file. Thus a good way to operate is

```
ed
e file
[editing session]
w
q
```

This way, you can simply say w from time to time, and be secure in the knowledge that if you got the file name right at the beginning, you are writing into the proper file each time.

You can find out at any time what file name *ed* is remembering by typing the *file* command f. In this example, if you typed

```
f
```

ed would reply

```
junk
```

Reading text from a file — the Read command "r"

Sometimes you want to read a file into the buffer without destroying anything that is already there. This is done by the *read* command r. The command

```
r junk
```

will read the file **junk** into the buffer; it adds it to the end of whatever is already in the buffer. So if you do a read after an edit:

```
e junk
r junk
```

the buffer will contain *two* copies of the text (six lines).

```
Now is the time
for all good men
to come to the aid of their party.
Now is the time
for all good men
to come to the aid of their party.
```

Like the w and e commands, r prints the number of characters read in, after the reading operation is complete.

Generally speaking, r is much less used than e.

Exercise 2:

Experiment with the e command — try reading and printing various files. You may get an error ?**name**, where **name** is the name of a file; this means that the file doesn't exist, typically because you spelled the file name wrong, or perhaps that you are not allowed to read or write it. Try alternately reading and appending to see that they work similarly. Verify that

```
ed filename
```

is exactly equivalent to

```
ed
e filename
```

What does

```
f filename
```

do?

Printing the contents of the buffer — the Print command "p"

To *print* or list the contents of the buffer (or parts of it) on the terminal, use the print command

```
p
```

The way this is done is as follows. Specify the lines where you want printing to begin and where you want it to end, separated by a comma, and followed by the letter p. Thus to print the first two lines of the buffer, for example, (that is, lines 1 through 2) say

```
1,2p      (starting line=1, ending line=2 p)
```

Ed will respond with

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

Suppose you want to print *all* the lines in the buffer. You could use 1,3p as above if you knew there were exactly 3 lines in the buffer. But in general, you don't know how many there are, so what do you use for the ending line number? *Ed* provides a shorthand symbol for "line number of last line in buffer" — the dollar sign $. Use it this way:

```
1,$p
```

This will print *all* the lines in the buffer (line 1 to last line.) If you want to stop the printing before it is finished, push the DEL or Delete key; *ed* will type

```
?
```

and wait for the next command.

To print the *last* line of the buffer, you could use

$,$p

but *ed* lets you abbreviate this to

$p

You can print any single line by typing the line number followed by a **p**. Thus

1p

produces the response

Now is the time

which is the first line of the buffer.

In fact, *ed* lets you abbreviate even further: you can print any single line by typing *just* the line number — no need to type the letter **p**. So if you say

$

ed will print the last line of the buffer.

You can also use **$** in combinations like

$−1,$p

which prints the last two lines of the buffer. This helps when you want to see how far you got in typing.

Exercise 3:

As before, create some text using the **a** command and experiment with the **p** command. You will find, for example, that you can't print line 0 or a line beyond the end of the buffer, and that attempts to print a buffer in reverse order by saying

3,1p

don't work.

The current line — "Dot" or "."

Suppose your buffer still contains the six lines as above, that you have just typed

1,3p

and *ed* has printed the three lines for you. Try typing just

p **(no line numbers)**

This will print

to come to the aid of their party.

which is the third line of the buffer. In fact it is the last (most recent) line that you have done anything with. (You just printed it!) You can repeat this **p** command without line numbers, and it will continue to print line 3.

The reason is that *ed* maintains a record of the last line that you did anything to (in this case, line 3, which you just printed) so that it can be used instead of an explicit line number. This most recent line is referred to by the shorthand symbol

. **(pronounced "dot").**

Dot is a line number in the same way that **$** is; it means exactly "the current line", or loosely, "the line you most recently did something to." You can use it in several ways — one possibility is to say

.,$p

This will print all the lines from (including) the current line to the end of the buffer. In our example these are lines 3 through 6.

Some commands change the value of dot, while others do not. The **p** command sets dot to the number of the last line printed; the last command will set both . and **$** to 6.

Dot is most useful when used in combinations like this one:

.+1 **(or equivalently, .+1p)**

This means "print the next line" and is a handy way to step slowly through a buffer. You can also say

.−1 **(or .−1p)**

which means "print the line *before* the current line." This enables you to go backwards if you wish. Another useful one is something like

.−3,.−1p

which prints the previous three lines.

Don't forget that all of these change the value of dot. You can find out what dot is at any time by typing

.=

Ed will respond by printing the value of dot.

Let's summarize some things about the **p** command and dot. Essentially **p** can be preceded by 0, 1, or 2 line numbers. If there is no line number given, it prints the "current line", the line that dot refers to. If there is one line number given (with or without the letter **p**), it prints that line (and dot is set there); and if there are two line numbers, it prints all the lines in that range (and sets dot to the last line printed.) If two line numbers are specified the first can't be bigger than the second (see Exercise 2.)

Typing a single return will cause printing of the next line — it's equivalent to **.+1p**. Try it. Try typing a **−**; you will find that it's equivalent to **.−1p**.

Deleting lines: the "d" command

Suppose you want to get rid of the three extra lines in the buffer. This is done by the *delete* command

```
d
```

Except that **d** deletes lines instead of printing them, its action is similar to that of **p**. The lines to be deleted are specified for **d** exactly as they are for **p**:

starting line, ending line **d**

Thus the command

```
4,$d
```

deletes lines 4 through the end. There are now three lines left, as you can check by using

```
1,$p
```

And notice that **$** now is line 3! Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to **$**.

Exercise 4:

Experiment with **a**, **e**, **r**, **w**, **p** and **d** until you are sure that you know what they do, and until you understand how dot, **$**, and line numbers are used.

If you are adventurous, try using line numbers with **a**, **r** and **w** as well. You will find that **a** will append lines *after* the line number that you specify (rather than after dot); that **r** reads a file in *after* the line number you specify (not necessarily at the end of the buffer); and that **w** will write out exactly the lines you specify, not necessarily the whole buffer. These variations are sometimes handy. For instance you can insert a file at the beginning of a buffer by saying

```
0r filename
```

and you can enter lines at the beginning of the buffer by saying

```
0a
. . . text . . .
.
```

Notice that **.w** is *very* different from

```
.
w
```

Modifying text: the Substitute command "s"

We are now ready to try one of the most important of all commands — the substitute command

```
s
```

This is the command that is used to change individual words or letters within a line or group of lines. It is what you use, for example, for correcting spelling mistakes and typing errors.

Suppose that by a typing error, line 1 says

```
Now is th time
```

— the *e* has been left off *the*. You can use **s** to fix this up as follows:

```
1s/th/the/
```

This says: "in line 1, substitute for the characters *th* the characters *the*." To verify that it works (*ed* will not print the result automatically) say

```
p
```

and get

```
Now is the time
```

which is what you wanted. Notice that dot must have been set to the line where the substitution took place, since the **p** command printed that line. Dot is always set this way with the **s** command.

The general way to use the substitute command is

starting-line, ending-line **s/** *change this/ to this/*

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in *all* the lines between *starting-line* and *ending-line*. Only the first occurrence on each line is changed, however. If you want to change *every* occurrence, see Exercise 5. The rules for line numbers are the same as those for **p**, except that dot is set to the last line changed. (But there is a trap for the unwary: if no substitution took place, dot is *not* changed. This causes an error **?** as a warning.)

Thus you can say

```
1,$s/speling/spelling/
```

and correct the first spelling mistake on each line in the text. (This is useful for people who are consistent misspellers!)

If no line numbers are given, the **s** command assumes we mean "make the substitution on line dot", so it changes things only on the current line. This leads to the very common sequence

```
s/something/something else/p
```

which makes some correction on the current line, and then prints it, to make sure it worked out right. If it didn't, you can try again. (Notice that there is a **p** on the same line as the **s** command. With few exceptions, **p** can follow any command; no other multi-command lines are legal.)

It's also legal to say

s/ . . . //

which means "change the first string of characters to "*nothing*", i.e., remove them. This is useful for deleting extra words in a line or removing extra letters from words. For instance, if you had

Nowxx is the time

you can say

s/xx//p

to get

Now is the time

Notice that // (two adjacent slashes) means "no characters", not a blank. There *is* a difference! (See below for another meaning of //.)

Exercise 5:

Experiment with the substitute command. See what happens if you substitute for some word on a line with several occurrences of that word. For example, do this:

a
the other side of the coin
.
s/the/on the/p

You will get

on the other side of the coin

A substitute command changes only the first occurrence of the first string. You can change all occurrences by adding a **g** (for "global") to the **s** command, like this:

s/ . . . / . . . /gp

Try other characters instead of slashes to delimit the two sets of characters in the **s** command — anything should work except blanks or tabs.

(If you get funny results using any of the characters

^ . $ [* \ &

read the section on "Special Characters".)

Context searching — "/ . . . /"

With the substitute command mastered, you can move on to another highly important idea of *ed* — context searching.

Suppose you have the original three line text in the buffer:

Now is the time
for all good men
to come to the aid of their party.

Suppose you want to find the line that contains *their* so you can change it to *the*. Now with only three lines in the buffer, it's pretty easy to keep track of what line the word *their* is on. But if the buffer contained several hundred lines, and you'd been making changes, deleting and rearranging lines, and so on, you would no longer really know what this line number would be. Context searching is simply a method of specifying the desired line, regardless of what its number is, by specifying some context on it.

The way to say "search for a line that contains this particular string of characters" is to type

/string of characters we want to find/

For example, the *ed* command

/their/

is a context search which is sufficient to find the desired line — it will locate the next occurrence of the characters between slashes ("their"). It also sets dot to that line and prints the line for verification:

to come to the aid of their party.

"Next occurrence" means that *ed* starts looking for the string at line **.+1**, searches to the end of the buffer, then continues at line 1 and searches to line dot. (That is, the search "wraps around" from **$** to 1.) It scans all the lines in the buffer until it either finds the desired line or gets back to dot again. If the given string of characters can't be found in any line, *ed* types the error message

?

Otherwise it prints the line it found.

You can do both the search for the desired line *and* a substitution all at once, like this:

/their/s/their/the/p

which will yield

to come to the aid of the party.

There were three parts to that last command: context search for the desired line, make the substitution, print the line.

The expression **/their/** is a context search expression. In their simplest form, all context search expressions are like this — a string of characters surrounded by slashes. Context searches are interchangeable with line numbers, so they can be used by themselves to find and print a desired line, or as line numbers for some other command, like **s**. They were used both ways in the examples above.

Suppose the buffer contains the three familiar lines

```
Now is the time
for all good men
to come to the aid of their party.
```

Then the *ed* line numbers

```
/Now/+1
/good/
/party/-1
```

are all context search expressions, and they all refer to the same line (line 2). To make a change in line 2, you could say

```
/Now/+1s/good/bad/
```

or

```
/good/s/good/bad/
```

or

```
/party/-1s/good/bad/
```

The choice is dictated only by convenience. You could print all three lines by, for instance

```
/Now/,/party/p
```

or

```
/Now/,/Now/+2p
```

or by any number of similar combinations. The first one of these might be better if you don't know how many lines are involved. (Of course, if there were only three lines in the buffer, you'd use

```
1,$p
```

but not if there were several hundred.)

The basic rule is: a context search expression is *the same as* a line number, so it can be used wherever a line number is needed.

Exercise 6:

Experiment with context searching. Try a body of text with several occurrences of the same string of characters, and scan through it using the same context search.

Try using context searches as line numbers for the substitute, print and delete commands. (They can also be used with r, w, and a.)

Try context searching using ?text? instead of /text/. This scans lines in the buffer in reverse order rather than normal. This is sometimes useful if you go too far while looking for some string of characters — it's an easy way to back up.

(If you get funny results with any of the characters

```
^ . $ [ * \ &
```

read the section on "Special Characters".)

Ed provides a shorthand for repeating a context search for the same string. For example, the *ed* line number

```
/string/
```

will find the next occurrence of **string**. It often happens that this is not the desired line, so the search must be repeated. This can be done by typing merely

```
//
```

This shorthand stands for "the most recently used context search expression." It can also be used as the first string of the substitute command, as in

```
/string1/s//string2/
```

which will find the next occurrence of **string1** and replace it by **string2**. This can save a lot of typing. Similarly

```
??
```

means "scan backwards for the same expression."

Change and Insert — "c" and "i"

This section discusses the *change* command

```
c
```

which is used to change or replace a group of one or more lines, and the *insert* command

```
i
```

which is used for inserting a group of one or more lines.

"Change", written as

```
c
```

is used to replace a number of lines with different lines, which are typed in at the terminal. For example, to change lines .+1 through $ to something else, type

```
.+1,$c
. . . type the lines of text you want here . . .
.
```

The lines you type between the c command and the . will take the place of the original lines between start line and end line. This is most useful in replacing a line or several lines which have errors in them.

If only one line is specified in the c command, then just that line is replaced. (You can type in as many replacement lines as you like.) Notice the use of . to end the input — this works just like the . in the append command

and must appear by itself on a new line. If no line number is given, line dot is replaced. The value of dot is set to the last line you typed in.

"Insert" is similar to append — for instance

```
/string/i
. . . type the lines to be inserted here . . .
.
```

will insert the given text *before* the next line that contains "string". The text between i and . is *inserted before* the specified line. If no line number is specified dot is used. Dot is set to the last line inserted.

Exercise 7:

"Change" is rather like a combination of delete followed by insert. Experiment to verify that

```
start, end d
i
. . . text . . .
.
```

is almost the same as

```
start, end c
. . . text . . .
.
```

These are not *precisely* the same if line $ gets deleted. Check this out. What is dot?

Experiment with a and i, to see that they are similar, but not the same. You will observe that

```
line-number a
. . . text . . .
.
```

appends *after* the given line, while

```
line-number i
. . . text . . .
.
```

inserts *before* it. Observe that if no line number is given, i inserts before line dot, while a appends after line dot.

Moving text around: the "m" command

The move command m is used for cutting and pasting — it lets you move a group of lines from one place to another in the buffer. Suppose you want to put the first three lines of the buffer at the end instead. You could do it by saying:

```
1,3w temp
$r temp
1,3d
```

(Do you see why?) but you can do it a lot easier with the m command:

```
1,3m$
```

The general case is

start line, end line m *after this line*

Notice that there is a third line to be specified — the place where the moved stuff gets put. Of course the lines to be moved can be specified by context searches; if you had

```
First paragraph
. . .
end of first paragraph.
Second paragraph
. . .
end of second paragraph.
```

you could reverse the two paragraphs like this:

```
/Second/,/end of second/m/First/-1
```

Notice the −1: the moved text goes *after* the line mentioned. Dot gets set to the last line moved.

The global commands "g" and "v"

The *global* command g is used to execute one or more *ed* commands on all those lines in the buffer that match some specified string. For example

```
g/peling/p
```

prints all lines that contain peling. More usefully,

```
g/peling/s//pelling/gp
```

makes the substitution everywhere on the line, then prints each corrected line. Compare this to

```
1,$s/peling/pelling/gp
```

which only prints the last line substituted. Another subtle difference is that the g command does not give a ? if peling is not found where the s command will.

There may be several commands (including a, c, i, r, w, but not g); in that case, every line except the last must end with a backslash \:

```
g/xxx/.-1s/abc/def/B
.+2s/ghi/jkl/B
.-2,.p
```

makes changes in the lines before and after each line that contains xxx, then prints all three lines.

The v command is the same as g, except that the commands are executed on every line that does *not* match the string following v:

```
v/ /d
```

deletes every line that does not contain a blank.

Special Characters

You may have noticed that things just don't work right when you used some characters like ., *, $, and others in context searches and the substitute command. The reason is rather complex, although the cure is simple. Basically, *ed* treats these characters as special, with special meanings. For instance, *in a context search or the first string of the substitute command only*, . means "any character," not a period, so

```
/x.y/
```

means "a line with an x, *any character*, and a y," *not* just "a line with an x, a period, and a y." A complete list of the special characters that can cause trouble is the following:

```
^  .  $  [  *  \
```

Warning: The backslash character \ is special to *ed*. For safety's sake, avoid it where possible. If you have to use one of the special characters in a substitute command, you can turn off its magic meaning temporarily by preceding it with the backslash. Thus

```
s/\\\.\*/backslash dot star/
```

will change \.* into "backslash dot star".

Here is a hurried synopsis of the other special characters. First, the circumflex ^ signifies the beginning of a line. Thus

```
/^string/
```

finds string only if it is at the beginning of a line: it will find

```
string
```

but not

```
the string...
```

The dollar-sign $ is just the opposite of the circumflex; it means the end of a line:

```
/string$/
```

will only find an occurrence of string that is at the end of some line. This implies, of course, that

```
/^string$/
```

will find only a line that contains just string, and

```
/^.$/
```

finds a line containing exactly one character.

The character ., as we mentioned above, matches anything:

```
/x.y/
```

matches any of

```
x+y
x-y
x y
x.y
```

This is useful in conjunction with *, which is a repetition character: a* is a shorthand for "any number of a's," so .* matches any number of anythings. This is used like this:

```
s/.*/stuff/
```

which changes an entire line, or

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

which deletes all characters in the line up to and including the last comma. (Since .* finds the longest possible match, this goes up to the last comma.)

[is used with] to form "character classes"; for example,

```
/[0123456789]/
```

matches any single digit — any one of the characters inside the braces will cause a match. This can be abbreviated to [0-9].

Finally, the & is another shorthand character — it is used only on the right-hand part of a substitute command where it means "whatever was matched on the left-hand side". It is used to save typing. Suppose the current line contained

```
Now is the time
```

and you wanted to put parentheses around it. You could just retype the line, but this is tedious. Or you could say

```
s/^/(/
s/$/)/
```

using your knowledge of ^ and $. But the easiest way uses the &:

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

This says "match the whole line, and replace it by itself surrounded by parentheses." The & can be used several times in a line; consider using

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

to produce

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

You don't have to match the whole line, of course: if the buffer contains

```
the end of the world
```

you could type

```
/world/s//& is at hand/
```

to produce

the end of the world is at hand

Observe this expression carefully, for it illustrates how to take advantage of *ed* to save typing. The string **/world/** found the desired line; the shorthand **//** found the same word in the line; and the **&** saves you from typing it again.

The **&** is a special character only within the replacement text of a substitute command, and has no special meaning elsewhere. You can turn off the special meaning of **&** by preceding it with a \:

s/ampersand/\&/

will convert the word "ampersand" into the literal symbol **&** in the current line.

Summary of Commands and Line Numbers

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

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

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

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

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

f: Print remembered filename. If a name follows **f** the remembered name will be set to it.

g: The command

g/---/commands

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

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

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

p: Print specified lines. If none specified, print line dot. A single line number is equivalent to *line-number* **p**. A single return prints **.+1**, the next line.

q: Quit *ed*. Wipes out all text in buffer if you give it twice in a row without first giving a **w** command.

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

s: The command

s/string1/string2/

substitutes the characters **string1** into **string2** in the specified lines. If no lines are specified, make the substitution in line dot. Dot is set to last line in which a substitution took place, which means that if no substitution took place, dot is not changed. **s** changes only the first occurrence of **string1** on a line; to change all of them, type a **g** after the final slash.

v: The command

v/---/commands

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

w: Write out buffer onto a file. Dot is not changed.

.=: Print value of dot. (**=** by itself prints the value of **$**.)

!: The line

!command-line

causes **command-line** to be executed as a UNIX command.

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

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

Advanced Editing on UNIX

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper is meant to help secretaries, typists and programmers to make effective use of the UNIX† facilities for preparing and editing text. It provides explanations and examples of

- special characters, line addressing and global commands in the editor **ed**;
- commands for "cut and paste" operations on files and parts of files, including the **mv**, **cp**, **cat** and **rm** commands, and the **r**, **w**, **m** and **t** commands of the editor;
- editing scripts and editor-based programs like **grep** and **sed**.

Although the treatment is aimed at non-programmers, new users with any background should find helpful hints on how to get their jobs done more easily.

August 4, 1978

†UNIX is a Trademark of Bell Laboratories.

Advanced Editing on UNIX

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

Although UNIX† provides remarkably effective tools for text editing, that by itself is no guarantee that everyone will automatically make the most effective use of them. In particular, people who are not computer specialists — typists, secretaries, casual users — often use the system less effectively than they might.

This document is intended as a sequel to *A Tutorial Introduction to the UNIX Text Editor* [1], providing explanations and examples of how to edit with less effort. (You should also be familiar with the material in *UNIX For Beginners* [2].) Further information on all commands discussed here can be found in *The UNIX Programmer's Manual* [3].

Examples are based on observations of users and the difficulties they encounter. Topics covered include special characters in searches and substitute commands, line addressing, the global commands, and line moving and copying. There are also brief discussions of effective use of related tools, like those for file manipulation, and those based on **ed**, like **grep** and **sed**.

A word of caution. There is only one way to learn to use something, and that is to *use* it. Reading a description is no substitute for trying something. A paper like this one should give you ideas about what to try, but until you actually try something, you will not learn it.

2. SPECIAL CHARACTERS

The editor **ed** is the primary interface to the system for many people, so it is worthwhile to know how to get the most out of **ed** for the least effort.

The next few sections will discuss shortcuts and labor-saving devices. Not all of these will be instantly useful to any one person, of course, but a few will be, and the others should give you ideas to store away for future use. And as always, until you try these things, they will remain theoretical knowledge, not something you have confidence in.

The List command 'l'

ed provides two commands for printing the contents of the lines you're editing. Most people are familiar with **p**, in combinations like

```
1,$p
```

to print all the lines you're editing, or

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

to change 'abc' to 'def' on the current line. Less familiar is the *list* command **l** (the letter 'l'), which gives slightly more information than **p**. In particular, **l** makes visible characters that are normally invisible, such as tabs and backspaces. If you list a line that contains some of these, **l** will print each tab as ➢ and each backspace as ≺. This makes it much easier to correct the sort of typing mistake that inserts extra spaces adjacent to tabs, or inserts a backspace followed by a space.

The **l** command also 'folds' long lines for printing — any line that exceeds 72 characters is printed on multiple lines; each printed line except the last is terminated by a backslash \, so you can tell it was folded. This is useful for printing long lines on short terminals.

Occasionally the **l** command will print in a line a string of numbers preceded by a backslash, such as \07 or \16. These combinations are used to make visible characters that normally don't print, like form feed or vertical tab or bell. Each such combination is a single character. When you see such characters, be wary — they may have surprising meanings when printed on some terminals. Often their presence means that your finger slipped while you were typing; you almost never want them.

The Substitute Command 's'

Most of the next few sections will be taken up with a discussion of the substitute command **s**. Since this is the command for changing the

†UNIX is a Trademark of Bell Laboratories.

contents of individual lines, it probably has the most complexity of any ed command, and the most potential for effective use.

As the simplest place to begin, recall the meaning of a trailing g after a substitute command. With

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

and

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

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

Either form of the s command can be followed by p or l to 'print' or 'list' (as described in the previous section) the contents of the line:

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

are all legal, and mean slightly different things. Make sure you know what the differences are.

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

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

changes the *first* occurrence of 'mispell' to 'misspell' on every line of the file. But

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

changes *every* occurrence in every line (and this is more likely to be what you wanted in this particular case).

You should also notice that if you add a p or l to the end of any of these substitute commands, only the last line that got changed will be printed, not all the lines. We will talk later about how to print all the lines that were modified.

The Undo Command 'u'

Occasionally you will make a substitution in a line, only to realize too late that it was a ghastly mistake. The 'undo' command u lets you 'undo' the last substitution: the last line that was substituted can be restored to its previous state by typing the command

```
u
```

The Metacharacter '.'

As you have undoubtedly noticed when you use ed, certain characters have unexpected meanings when they occur in the left side of a substitute command, or in a search for a particular line. In the next several sections, we will talk about these special characters, which are often called 'metacharacters'.

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

```
/x.y/
```

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

```
x+y
x−y
x□y
x.y
```

and so on. (We will use □ to stand for a space whenever we need to make it visible.)

Since '.' matches a single character, that gives you a way to deal with funny characters printed by l. Suppose you have a line that, when printed with the l command, appears as

```
.... th\07is ....
```

and you want to get rid of the \07 (which represents the bell character, by the way).

The most obvious solution is to try

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

but this will fail. (Try it.) The brute force solution, which most people would now take, is to re-type the entire line. This is guaranteed, and is actually quite a reasonable tactic if the line in question isn't too big, but for a very long line, re-typing is a bore. This is where the metacharacter '.' comes in handy. Since '\07' really represents a single character, if we say

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

the job is done. The '.' matches the mysterious character between the 'h' and the 'i', *whatever it is.*

Bear in mind that since '.' matches any single character, the command

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

converts the first character on a line into a ',', which very often is not what you intended.

As is true of many characters in ed, the '.' has several meanings, depending on its context. This line shows all three:

```
.s/./
```

The first '.' is a line number, the number of the line we are editing, which is called 'line dot'. (We will discuss line dot more in Section 3.) The second '.' is a metacharacter that matches any single character on that line. The third '.' is the only one that really is an honest literal period. On the *right* side of a substitution, '.' is not special. If you apply this command to the line

```
Now is the time.
```

the result will be

```
.ow is the time.
```

which is probably not what you intended.

The Backslash '\'

Since a period means 'any character', the question naturally arises of what to do when you really want a period. For example, how do you convert the line

```
Now is the time.
```

into

```
Now is the time?
```

The backslash '\' does the job. A backslash turns off any special meaning that the next character might have; in particular, '\.' converts the '.' from a 'match anything' into a period, so you can use it to replace the period in

```
Now is the time.
```

like this:

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

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

The backslash can also be used when searching for lines that contain a special character. Suppose you are looking for a line that contains

```
.PP
```

The search

```
/.PP/
```

isn't adequate, for it will find a line like

```
THE APPLICATION OF ...
```

because the '.' matches the letter 'A'. But if you say

```
/\.PP/
```

you will find only lines that contain '.PP'.

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

```
/\/
```

won't work, because the '\' isn't a literal '\', but instead means that the second '/' no longer delimits the search. But by preceding a backslash with another one, you can search for a literal backslash. Thus

```
/\\/
```

does work. Similarly, you can search for a forward slash '/' with

```
/\//
```

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

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

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

into the line

```
\x\y
```

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

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

A couple of miscellaneous notes about backslashes and special characters. First, you can use any character to delimit the pieces of an s command: there is nothing sacred about slashes. (But you must use slashes for context searching.) For instance, in a line that contains a lot of slashes already, like

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

you could use a colon as the delimiter — to delete all the slashes, type

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

Second, if # and @ are your character erase and line kill characters, you have to type \# and \@; this is true whether you're talking to ed or any other program.

When you are adding text with a or i or c, backslash is not special, and you should only put in one backslash for each one you really want.

The Dollar Sign '$'

The next metacharacter, the '$', stands for 'the end of the line'. As its most obvious use, suppose you have the line

Now is the

and you wish to add the word 'time' to the end. Use the $ like this:

s/$/␣time/

to get

Now is the time

Notice that a space is needed before 'time' in the substitute command, or you will get

Now is thetime

As another example, replace the second comma in the following line with a period without altering the first:

Now is the time, for all good men,

The command needed is

s/,$/./

The $ sign here provides context to make specific which comma we mean. Without it, of course, the s command would operate on the first comma to produce

Now is the time. for all good men,

As another example, to convert

Now is the time.

into

Now is the time?

as we did earlier, we can use

s/.$/?/

Like '.', the '$' has multiple meanings depending on context. In the line

$s/$/$/

the first '$' refers to the last line of the file, the second refers to the end of that line, and the third is a literal dollar sign, to be added to that line.

The Circumflex '^'

The circumflex (or hat or caret) '^' stands for the beginning of the line. For example, suppose you are looking for a line that begins with 'the'. If you simply say

/the/

you will in all likelihood find several lines that contain 'the' in the middle before arriving at the one you want. But with

/^the/

you narrow the context, and thus arrive at the desired one more easily.

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

s/^/␣/

places a space at the beginning of the current line.

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

.PP

you can use the command

/^\.PP$/

The Star '*'

Suppose you have a line that looks like this:

text x y *text*

where *text* stands for lots of text, and there are some indeterminate number of spaces between the x and the y. Suppose the job is to replace all the spaces between x and y by a single space. The line is too long to retype, and there are too many spaces to count. What now?

This is where the metacharacter '*' comes in handy. A character followed by a star stands for as many consecutive occurrences of that character as possible. To refer to all the spaces at once, say

s/x␣*y/x␣y/

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

The star can be used with any character, not just space. If the original example was instead

text x---------y *text*

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

s/x-*y/x␣y/

Finally, suppose that the line was

text x.....y *text*

Can you see what trap lies in wait for the unwary? If you blindly type

s/x.*y/x␣y/

what will happen? The answer, naturally, is that it depends. If there are no other x's or y's on the line, then everything works, but it's blind luck, not good management. Remember that '.' matches *any* single character? Then '.*' matches as many single characters as possible, and unless

you're careful, it can eat up a lot more of the line than you expected. If the line was, for example, like this:

text x *text* x.....y *text* y *text*

then saying

s/x.*y/x␣y/

will take everything from the *first* 'x' to the *last* 'y', which, in this example, is undoubtedly more than you wanted.

The solution, of course, is to turn off the special meaning of '.' with '\.':

s/x\.*y/x␣y/

Now everything works, for '\.*' means 'as many *periods* as possible'.

There are times when the pattern '.*' is exactly what you want. For example, to change

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

into

Now is the time.

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

s/␣for.*/./

There are a couple of additional pitfalls associated with '*' that you should be aware of. Most notable is the fact that 'as many as possible' means *zero* or more. The fact that zero is a legitimate possibility is sometimes rather surprising. For example, if our line contained

text xy *text* x y *text*

and we said

s/x␣*y/x␣y/

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

The way around this, if it matters, is to specify a pattern like

/x␣␣*y/

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

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

s/x*/y/g

when applied to the line

abcdef

produces

yaybycydyeyfy

which is almost certainly not what was intended. The reason for this behavior is that zero is a legal number of matches, and there are no x's at the beginning of the line (so that gets converted into a 'y'), nor between the 'a' and the 'b' (so that gets converted into a 'y'), nor ... and so on. Make sure you really want zero matches; if not, in this case write

s/xx*/y/g

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

The Brackets '[ ]'

Suppose that you want to delete any numbers that appear at the beginning of all lines of a file. You might first think of trying a series of commands like

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

and so on, but this is clearly going to take forever if the numbers are at all long. Unless you want to repeat the commands over and over until finally all numbers are gone, you must get all the digits on one pass. This is the purpose of the brackets [and].

The construction

[0123456789]

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

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

deletes all digits from the beginning of all lines.

Any characters can appear within a character class, and just to confuse the issue there are essentially no special characters inside the brackets: even the backslash doesn't have a special meaning. To search for special characters, for example, you can say

/[.\$^[]/

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

It's a nuisance to have to spell out the digits, so you can abbreviate them as [0-9]; similarly, [a-z] stands for the lower case letters, and [A-Z] for upper case.

As a final frill on character classes, you can

specify a class that means 'none of the following characters'. This is done by beginning the class with a '^':

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

stands for 'any character *except* a digit'. Thus you might find the first line that doesn't begin with a tab or space by a search like

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

Within a character class, the circumflex has a special meaning only if it occurs at the beginning. Just to convince yourself, verify that

```
/^[^^]/
```

finds a line that doesn't begin with a circumflex.

The Ampersand '&'

The ampersand '&' is used primarily to save typing. Suppose you have the line

```
Now is the time
```

and you want to make it

```
Now is the best time
```

Of course you can always say

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

but it seems silly to have to repeat the 'the'. The '&' is used to eliminate the repetition. On the *right* side of a substitute, the ampersand means 'whatever was just matched', so you can say

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

and the '&' will stand for 'the'. Of course this isn't much of a saving if the thing matched is just 'the', but if it is something truly long or awful, or if it is something like '.*' which matches a lot of text, you can save some tedious typing. There is also much less chance of making a typing error in the replacement text. For example, to parenthesize a line, regardless of its length,

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

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

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

makes

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

and

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

converts the original line into

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

To get a literal ampersand, naturally the backslash is used to turn off the special meaning:

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

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

Substituting Newlines

ed provides a facility for splitting a single line into two or more shorter lines by 'substituting in a newline'. As the simplest example, suppose a line has gotten unmanageably long because of editing (or merely because it was unwisely typed). If it looks like

```
text  xy  text
```

you can break it between the 'x' and the 'y' like this:

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

This is actually a single command, although it is typed on two lines. Bearing in mind that '\' turns off special meanings, it seems relatively intuitive that a '\' at the end of a line would make the newline there no longer special.

You can in fact make a single line into several lines with this same mechanism. As a large example, consider underlining the word 'very' in a long line by splitting 'very' onto a separate line, and preceding it by the roff or nroff formatting command '.ul'.

```
text a very big text
```

The command

```
s/␣very␣/\
.ul\
very\
/
```

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

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

Joining Lines

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

```
Now is
␣the time
```

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

then the command

```
j
```

joins them together. No blanks are added, which is why we carefully showed a blank at the beginning of the second line.

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

```
1,$jp
```

joins all the lines into one big one and prints it. (More on line numbers in Section 3.)

Rearranging a Line with \(... \)

(This section should be skipped on first reading.) Recall that '&' is a shorthand that stands for whatever was matched by the left side of an s command. In much the same way you can capture separate pieces of what was matched: the only difference is that you have to specify on the left side just what pieces you're interested in.

Suppose, for instance, that you have a file of lines that consist of names in the form

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

and so on, and you want the initials to precede the name, as in

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

It is possible to do this with a series of editing commands, but it is tedious and error-prone. (It is instructive to figure out how it is done, though.)

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

The command

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

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

Of course, with any editing sequence this complicated, it's foolhardy to simply run it and hope. The global commands g and v discussed in section 4 provide a way for you to print exactly those lines which were affected by the substitute command, and thus verify that it did what you wanted in all cases.

3. LINE ADDRESSING IN THE EDITOR

The next general area we will discuss is that of line addressing in ed, that is, how you specify what lines are to be affected by editing commands. We have already used constructions like

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

to specify a change on all lines. And most users are long since familiar with using a single newline (or return) to print the next line, and with

```
/thing/
```

to find a line that contains 'thing'. Less familiar, surprisingly enough, is the use of

```
?thing?
```

to scan *backwards* for the previous occurrence of 'thing'. This is especially handy when you realize that the thing you want to operate on is back up the page from where you are currently editing.

The slash and question mark are the only characters you can use to delimit a context search, though you can use essentially any character in a substitute command.

Address Arithmetic

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

```
$-1
```

is a command to print the next to last line of the current file (that is, one line before line '$'). For example, to recall how far you got in a previous editing session,

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

prints the last six lines. (Be sure you understand why it's six, not five.) If there aren't six, of course, you'll get an error message.

As another example,

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

prints from three lines before where you are now (at line dot) to three lines after, thus giving you a bit of context. By the way, the '+' can be omitted:

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

is absolutely identical in meaning.

Another area in which you can save typing effort in specifying lines is to use '−' and '+' as line numbers by themselves.

```
−
```

by itself is a command to move back up one line in the file. In fact, you can string several minus signs together to move back up that many lines:

```
−−−
```

moves up three lines, as does '−3'. Thus

```
−3,+3p
```

is also identical to the examples above.

Since '−' is shorter than '.−1', constructions like

```
−,.s/bad/good/
```

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

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

```
/thing/−−
```

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

Repeated Searches

Suppose you ask for the search

```
/horrible thing/
```

and when the line is printed you discover that it isn't the horrible thing that you wanted, so it is necessary to repeat the search again. You don't have to re-type the search, for the construction

```
//
```

is a shorthand for 'the previous thing that was searched for', whatever it was. This can be repeated as many times as necessary. You can also go backwards:

```
??
```

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

Not only can you repeat the search, but you can use '//' as the left side of a substitute command, to mean 'the most recent pattern'.

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

To go backwards and change a line, say

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

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

```
//s//&␣&/p
```

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

Default Line Numbers and the Value of Dot

One of the most effective ways to speed up your editing is always to know what lines will be affected by a command if you don't specify the lines it is to act on, and on what line you will be positioned (i.e., the value of dot) when a command finishes. If you can edit without specifying unnecessary line numbers, you can save a lot of typing.

As the most obvious example, if you issue a search command like

```
/thing/
```

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

What happens if there was no 'thing'? Then you are left right where you were — dot is unchanged. This is also true if you were sitting on the only 'thing' when you issued the command. The same rules hold for searches that use '?...?'; the only difference is the direction in which you search.

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

The line-changing commands **a**, **c** and **i** by default all affect the current line — if you give no line number with them, **a** appends text after the current line, **c** changes the current line, and **i** inserts text before the current line.

a, **c**, and **i** behave identically in one respect — when you stop appending, changing or inserting, dot points at the last line entered. This is exactly what you want for typing and editing on the fly. For example, you can say

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

without specifying any line number for the sub-

stitute command or for the second append command. Or you can say

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

You should experiment to determine what happens if you add *no* lines with **a**, **c** or **i**.

The **r** command will read a file into the text being edited, either at the end if you give no address, or after the specified line if you do. In either case, dot points at the last line read in. Remember that you can even say **0r** to read a file in at the beginning of the text. (You can also say **0a** or **1i** to start adding text at the beginning.)

The **w** command writes out the entire file. If you precede the command by one line number, that line is written, while if you precede it by two line numbers, that range of lines is written. The **w** command does *not* change dot: the current line remains the same, regardless of what lines are written. This is true even if you say something like

```
/^\.AB/,/^\.AE/w abstract
```

which involves a context search.

Since the **w** command is so easy to use, you should save what you are editing regularly as you go along just in case the system crashes, or in case you do something foolish, like clobbering what you're editing.

The least intuitive behavior, in a sense, is that of the **s** command. The rule is simple — you are left sitting on the last line that got changed. If there were no changes, then dot is unchanged.

To illustrate, suppose that there are three lines in the buffer, and you are sitting on the middle one:

```
x1
x2
x3
```

Then the command

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

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

```
x1
y2
y3
```

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

Semicolon ';'

Searches with '/.../' and '?...?' start at the current line and move forward or backward respectively until they either find the pattern or get back to the current line. Sometimes this is not what is wanted. Suppose, for example, that the buffer contains lines like this:

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

Starting at line 1, one would expect that the command

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

prints all the lines from the 'ab' to the 'bc' inclusive. Actually this is not what happens. *Both* searches (for 'a' and for 'b') start from the same point, and thus they both find the line that contains 'ab'. The result is to print a single line. Worse, if there had been a line with a 'b' in it before the 'ab' line, then the print command would be in error, since the second line number would be less than the first, and it is illegal to try to print lines in reverse order.

This is because the comma separator for line numbers doesn't set dot as each address is processed; each search starts from the same place. In **ed**, the semicolon ';' can be used just like comma, with the single difference that use of a semicolon forces dot to be set at that point as the line numbers are being evaluated. In effect, the semicolon 'moves' dot. Thus in our example above, the command

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

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

This property is most often useful in a very simple situation. Suppose you want to find the *second* occurrence of 'thing'. You could say

```
/thing/
//
```

but this prints the first occurrence as well as the

second, and is a nuisance when you know very well that it is only the second one you're interested in. The solution is to say

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

This says to find the first occurrence of 'thing', set dot to that line, then find the second and print only that.

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

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

Printing the third or fourth or ... in either direction is left as an exercise.

Finally, bear in mind that if you want to find the first occurrence of something in a file, starting at an arbitrary place within the file, it is not sufficient to say

```
1;/thing/
```

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

```
0;/thing/
```

(one of the few places where 0 is a legal line number), for this starts the search at line 1.

Interrupting the Editor

As a final note on what dot gets set to, you should be aware that if you hit the interrupt or delete or rubout or break key while ed is doing a command, things are put back together again and your state is restored as much as possible to what it was before the command began. Naturally, some changes are irrevocable — if you are reading or writing a file or making substitutions or deleting lines, these will be stopped in some clean but unpredictable state in the middle (which is why it is not usually wise to stop them). Dot may or may not be changed.

Printing is more clear cut. Dot is not changed until the printing is done. Thus if you print until you see an interesting line, then hit delete, you are *not* sitting on that line or even near it. Dot is left where it was when the p command was started.

4. GLOBAL COMMANDS

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

As the simplest example, the command

```
g/UNIX/p
```

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

As another example, then,

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

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

The v command is identical to g, except that it operates on those line that do *not* contain an occurrence of the pattern. (Don't look too hard for mnemonic significance to the letter 'v'.) So

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

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

The command that follows g or v can be anything:

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

deletes all lines that begin with '.', and

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

deletes all empty lines.

Probably the most useful command that can follow a global is the substitute command, for this can be used to make a change and print each affected line for verification. For example, we could change the word 'Unix' to 'UNIX' everywhere, and verify that it really worked, with

```
g/Unix/s//UNIX/gp
```

Notice that we used '//' in the substitute command to mean 'the previous pattern', in this case, 'Unix'. The p command is done on every line that matches the pattern, not just those on which a substitution took place.

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

```
g/^\.PP/+
```

prints the line that follows each '.PP' command (the signal for a new paragraph in some formatting packages). Remember that '+' means 'one line past dot'. And

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

searches for each line that contains 'topic', scans backwards until it finds a line that begins '.SH' (a section heading) and prints the line that follows that, thus showing the section headings

under which 'topic' is mentioned. Finally,

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

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

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

Multi-line Global Commands

It is possible to do more than one command under the control of a global command, although the syntax for expressing the operation is not especially natural or pleasant. As an example, suppose the task is to change 'x' to 'y' and 'a' to 'b' on all lines that contain 'thing'. Then

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

is sufficient. The '\' signals the g command that the set of commands continues on the next line; it terminates on the first line that does not end with '\'. (As a minor blemish, you can't use a substitute command to insert a newline within a g command.)

You should watch out for this problem: the command

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

does *not* work as you expect. The remembered pattern is the last pattern that was actually executed, so sometimes it will be 'x' (as expected), and sometimes it will be 'a' (not expected). You must spell it out, like this:

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

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

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

There is no need for a final line containing a '.' to terminate the i command, unless there are further commands being done under the global. On the other hand, it does no harm to put it in either.

5. CUT AND PASTE WITH UNIX COMMANDS

One editing area in which non-programmers seem not very confident is in what might be called 'cut and paste' operations — changing the name of a file, making a copy of a file somewhere else, moving a few lines from one place to another in a file, inserting one file in the middle of another, splitting a file into pieces, and splicing two or more files together.

Yet most of these operations are actually quite easy, if you keep your wits about you and go cautiously. The next several sections talk about cut and paste. We will begin with the UNIX commands for moving entire files around, then discuss ed commands for operating on pieces of files.

Changing the Name of a File

You have a file named 'memo' and you want it to be called 'paper' instead. How is it done?

The UNIX program that renames files is called mv (for 'move'); it 'moves' the file from one name to another, like this:

```
mv memo paper
```

That's all there is to it: mv from the old name to the new name.

```
mv oldname newname
```

Warning: if there is already a file around with the new name, its present contents will be silently clobbered by the information from the other file. The one exception is that you can't move a file to itself —

```
mv x x
```

is illegal.

Making a Copy of a File

Sometimes what you want is a copy of a file — an entirely fresh version. This might be because you want to work on a file, and yet save a copy in case something gets fouled up, or just because you're paranoid.

In any case, the way to do it is with the cp command. (cp stands for 'copy'; the system is big on short command names, which are appreciated by heavy users, but sometimes a strain for novices.) Suppose you have a file called 'good' and you want to save a copy before you make some dramatic editing changes. Choose a name — 'savegood' might be acceptable — then type

```
cp good savegood
```

This copies 'good' onto 'savegood', and you now

have two identical copies of the file 'good'. (If 'savegood' previously contained something, it gets overwritten.)

Now if you decide at some time that you want to get back to the original state of 'good', you can say

```
mv savegood good
```

(if you're not interested in 'savegood' any more), or

```
cp savegood good
```

if you still want to retain a safe copy.

In summary, **mv** just renames a file; **cp** makes a duplicate copy. Both of them clobber the 'target' file if it already exists, so you had better be sure that's what you want to do *before* you do it.

Removing a File

If you decide you are really done with a file forever, you can remove it with the **rm** command:

```
rm savegood
```

throws away (irrevocably) the file called 'savegood'.

Putting Two or More Files Together

The next step is the familiar one of collecting two or more files into one big one. This will be needed, for example, when the author of a paper decides that several sections need to be combined into one. There are several ways to do it, of which the cleanest, once you get used to it, is a program called **cat**. (Not *all* programs have two-letter names.) **cat** is short for 'concatenate', which is exactly what we want to do.

Suppose the job is to combine the files 'file1' and 'file2' into a single file called 'bigfile'. If you say

```
cat file
```

the contents of 'file' will get printed on your terminal. If you say

```
cat file1 file2
```

the contents of 'file1' and then the contents of 'file2' will *both* be printed on your terminal, in that order. So **cat** combines the files, all right, but it's not much help to print them on the terminal — we want them in 'bigfile'.

Fortunately, there is a way. You can tell the system that instead of printing on your terminal, you want the same information put in a file. The way to do it is to add to the command line the character > and the name of the file where you want the output to go. Then you can say

```
cat file1 file2 >bigfile
```

and the job is done. (As with **cp** and **mv**, you're putting something into 'bigfile', and anything that was already there is destroyed.)

This ability to 'capture' the output of a program is one of the most useful aspects of the system. Fortunately it's not limited to the **cat** program — you can use it with *any* program that prints on your terminal. We'll see some more uses for it in a moment.

Naturally, you can combine several files, not just two:

```
cat file1 file2 file3 ... >bigfile
```

collects a whole bunch.

Question: is there any difference between

```
cp good savegood
```

and

```
cat good >savegood
```

Answer: for most purposes, no. You might reasonably ask why there are two programs in that case, since **cat** is obviously all you need. The answer is that **cp** will do some other things as well, which you can investigate for yourself by reading the manual. For now we'll stick to simple usages.

Adding Something to the End of a File

Sometimes you want to add one file to the end of another. We have enough building blocks now that you can do it; in fact before reading further it would be valuable if you figured out how. To be specific, how would you use **cp**, **mv** and/or **cat** to add the file 'good1' to the end of the file 'good'?

You could try

```
cat good good1 >temp
mv temp good
```

which is probably most direct. You should also understand why

```
cat good good1 >good
```

doesn't work. (Don't practice with a good 'good'!)

The easy way is to use a variant of >, called >>. In fact, >> is identical to > except that instead of clobbering the old file, it simply tacks stuff on at the end. Thus you could say

```
cat good1 >>good
```

and 'good1' is added to the end of 'good'. (And

if 'good' didn't exist, this makes a copy of 'good1' called 'good'.)

6. CUT AND PASTE WITH THE EDITOR

Now we move on to manipulating pieces of files — individual lines or groups of lines. This is another area where new users seem unsure of themselves.

Filenames

The first step is to ensure that you know the **ed** commands for reading and writing files. Of course you can't go very far without knowing **r** and **w**. Equally useful, but less well known, is the 'edit' command **e**. Within **ed**, the command

```
e newfile
```

says 'I want to edit a new file called *newfile*, without leaving the editor.' The **e** command discards whatever you're currently working on and starts over on *newfile*. It's exactly the same as if you had quit with the **q** command, then re-entered **ed** with a new file name, except that if you have a pattern remembered, then a command like **//** will still work.

If you enter **ed** with the command

```
ed file
```

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

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

(and so on) does a series of edits on various files without ever leaving **ed** and without typing the name of any file more than once. (As an aside, if you examine the sequence of commands here, you can see why many UNIX systems use **e** as a synonym for **ed**.)

You can find out the remembered file name at any time with the **f** command; just type **f** without a file name. You can also change the name of the remembered file name with **f**; a useful sequence is

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

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

Inserting One File into Another

Suppose you have a file called 'memo', and you want the file called 'table' to be inserted just after the reference to Table 1. That is, in 'memo' somewhere is a line that says

```
Table 1 shows that ...
```

and the data contained in 'table' has to go there, probably so it will be formatted properly by **nroff** or **troff**. Now what?

This one is easy. Edit 'memo', find 'Table 1', and add the file 'table' right there:

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

The critical line is the last one. As we said earlier, the **r** command reads a file; here you asked for it to be read in right after line dot. An **r** command without any address adds lines at the end, so it is the same as **$r**.

Writing out Part of a File

The other side of the coin is writing out part of the document you're editing. For example, maybe you want to split out into a separate file that table from the previous example, so it can be formatted and tested separately. Suppose that in the file being edited we have

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

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

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

and the job is done. If you are confident, you can do it all at once with

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

The point is that the **w** command can write out a group of lines, instead of the whole file. In fact, you can write out a single line if you like; just give one line number instead of two. For example, if you have just typed a horribly complicated line and you know that it (or something like it) is going to be needed later, then save it — don't re-type it. In the editor, say

```
a
...lots of stuff...
...horrible line...
.
.w temp
a
...more stuff...
.
.r temp
a
...more stuff...
.
```

This last example is worth studying, to be sure you appreciate what's going on.

Moving Lines Around

Suppose you want to move a paragraph from its present position in a paper to the end. How would you do it? As a concrete example, suppose each paragraph in the paper begins with the formatting command '.PP'. Think about it and write down the details before reading on.

The brute force way (not necessarily bad) is to write the paragraph onto a temporary file, delete it from its current position, then read in the temporary file at the end. Assuming that you are sitting on the '.PP' command that begins the paragraph, this is the sequence of commands:

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

That is, from where you are now ('.') until one line before the next '.PP' ('/^\.PP/-') write onto 'temp'. Then delete the same lines. Finally, read 'temp' at the end.

As we said, that's the brute force way. The easier way (often) is to use the *move* command m that ed provides — it lets you do the whole set of operations at one crack, without any temporary file.

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

```
line1, line2 m line3
```

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

Suppose again that you're sitting at the first line of the paragraph. Then you can say

```
../^\.PP/-m$
```

That's all.

As another example of a frequent operation, you can reverse the order of two adjacent lines by moving the first one to after the second. Suppose that you are positioned at the first. Then

```
m+
```

does it. It says to move line dot to after one line after line dot. If you are positioned on the second line,

```
m--
```

does the interchange.

As you can see, the m command is more succinct and direct than writing, deleting and rereading. When is brute force better anyway? This is a matter of personal taste — do what you have most confidence in. The main difficulty with the m command is that if you use patterns to specify both the lines you are moving and the target, you have to take care that you specify them properly, or you may well not move the lines you thought you did. The result of a botched m command can be a ghastly mess. Doing the job a step at a time makes it easier for you to verify at each step that you accomplished what you wanted to. It's also a good idea to issue a w command before doing anything complicated; then if you goof, it's easy to back up to where you were.

Marks

ed provides a facility for marking a line with a particular name so you can later reference it by name regardless of its actual line number. This can be handy for moving lines, and for keeping track of them as they move. The *mark* command is k; the command

```
kx
```

marks the current line with the name 'x'. If a line number precedes the k, that line is marked. (The mark name must be a single lower case letter.) Now you can refer to the marked line with the address

```
'x
```

Marks are most useful for moving things around. Find the first line of the block to be moved, and mark it with *'a*. Then find the last line and mark it with *'b*. Now position yourself at the place where the stuff is to go and say

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

Bear in mind that only one line can have a particular mark name associated with it at any given time.

Copying Lines

We mentioned earlier the idea of saving a line that was hard to type or used often, so as to cut down on typing time. Of course this could be more than one line; then the saving is presumably even greater.

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

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

```
1,$t$
```

duplicates the entire contents that you are editing. A more common use for t is for creating a series of lines that differ only slightly. For example, you can say

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

and so on.

The Temporary Escape '!'

Sometimes it is convenient to be able to temporarily escape from the editor to do some other UNIX command, perhaps one of the file copy or move commands discussed in section 5, without leaving the editor. The 'escape' command ! provides a way to do this.

If you say

```
!any UNIX command
```

your current editing state is suspended, and the UNIX command you asked for is executed. When the command finishes, **ed** will signal you by printing another !; at that point you can resume editing.

You can really do *any* UNIX command, including another **ed**. (This is quite common, in fact.) In this case, you can even do another !.

7. SUPPORTING TOOLS

There are several tools and techniques that go along with the editor, all of which are relatively easy once you know how **ed** works, because they are all based on the editor. In this section we will give some fairly cursory examples of these tools, more to indicate their existence than to provide a complete tutorial. More information on each can be found in [3].

Grep

Sometimes you want to find all occurrences of some word or pattern in a set of files, to edit them or perhaps just to verify their presence or absence. It may be possible to edit each file separately and look for the pattern of interest, but if there are many files this can get very tedious, and if the files are really big, it may be impossible because of limits in **ed**.

The program **grep** was invented to get around these limitations. The search patterns that we have described in the paper are often called 'regular expressions', and 'grep' stands for

```
g/re/p
```

That describes exactly what **grep** does — it prints every line in a set of files that contains a particular pattern. Thus

```
grep 'thing' file1 file2 file3 ...
```

finds 'thing' wherever it occurs in any of the files 'file1', 'file2', etc. **grep** also indicates the file in which the line was found, so you can later edit it if you like.

The pattern represented by 'thing' can be any pattern you can use in the editor, since **grep** and **ed** use exactly the same mechanism for pattern searching. It is wisest always to enclose the pattern in the single quotes '...' if it contains any non-alphabetic characters, since many such characters also mean something special to the UNIX command interpreter (the 'shell'). If you don't quote them, the command interpreter will try to interpret them before **grep** gets a chance.

There is also a way to find lines that *don't* contain a pattern:

```
grep -v 'thing' file1 file2 ...
```

finds all lines that don't contains 'thing'. The **-v** must occur in the position shown. Given **grep** and **grep -v**, it is possible to do things like selecting all lines that contain some combination of patterns. For example, to get all lines that contain 'x' but not 'y':

```
grep x file... | grep -v y
```

(The notation | is a 'pipe', which causes the output of the first command to be used as input to the second command; see [2].)

Editing Scripts

If a fairly complicated set of editing operations is to be done on a whole set of files, the easiest thing to do is to make up a 'script', i.e., a file that contains the operations you want to perform, then apply this script to each file in turn.

For example, suppose you want to change every 'Unix' to 'UNIX' and every 'Gcos' to 'GCOS' in a large number of files. Then put into the file 'script' the lines

```
g/Unix/s//UNIX/g
g/Gcos/s//GCOS/g
w
q
```

Now you can say

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

This causes **ed** to take its commands from the prepared script. Notice that the whole job has to be planned in advance.

And of course by using the UNIX command interpreter, you can cycle through a set of files automatically, with varying degrees of ease.

Sed

sed ('stream editor') is a version of the editor with restricted capabilities but which is capable of processing unlimited amounts of input. Basically **sed** copies its input to its output, applying one or more editing commands to each line of input.

As an example, suppose that we want to do the 'Unix' to 'UNIX' part of the example given above, but without rewriting the files. Then the command

```
sed 's/Unix/UNIX/g' file1 file2 ...
```

applies the command 's/Unix/UNIX/g' to all lines from 'file1', 'file2', etc., and copies all lines to the output. The advantage of using **sed** in such a case is that it can be used with input too large for **ed** to handle. All the output can be collected in one place, either in a file or perhaps piped into another program.

If the editing transformation is so complicated that more than one editing command is needed, commands can be supplied from a file, or on the command line, with a slightly more complex syntax. To take commands from a file, for example,

```
sed -f cmdfile input-files...
```

sed has further capabilities, including conditional testing and branching, which we cannot go into here.

Acknowledgement

I am grateful to Ted Dolotta for his careful reading and valuable suggestions.

References

[1] Brian W. Kernighan, *A Tutorial Introduction to the UNIX Text Editor*. Bell Laboratories internal memorandum.

[2] Brian W. Kernighan, *UNIX For Beginners*. Bell Laboratories internal memorandum.

[3] Ken L. Thompson and Dennis M. Ritchie, *The UNIX Programmer's Manual*. Bell Laboratories.

Ex Reference Manual

William Joy

Revised for Versions 3.5 (VAX UNIX) and 2.13 (PDP UNIX) by
Mark Horton

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

ABSTRACT

Ex is a line oriented text editor which supports both command and display oriented editing. This reference manual describes the command oriented part of *ex*; the display editing features of *ex* are described in *An Introduction to Display Editing with Vi*. Other documents about the editor include the introduction *Edit: A Tutorial*, the *Ex/Edit Command Summary*, and a *Vi Quick Reference* card.

Computer Science
September 1980

Computing Services Notes

This manual documents both versions 2.13 and 3.5 of the text editor *ex*. Version 2.13 is currently supported on the PDP UNIX Systems operated by Computer Facilities and Operations (CFO); version 3.6 is currently supported on the CFO VAX UNIX system. For VAX UNIX users, the changes from version 3.5 to 3.6 are listed in an appendix to this manual. Also, a cumulative list of changes to the editor from version to version is maintained online on all UNIX systems; to retrieve the information type **help ex news** on PDP UNIX and cat **/usr/news/ex** on VAX UNIX.

A small number of features available with version 3.5 are not found in version 2.13. In most cases, the manual documents these differences with footnotes stating "Version 3 only." But there are still some instances in which differences are not noted, or are noted ambiguously. Here is a complete list of the editor's commands and options available in version 3.5 but not in 2.13:

Ex Commands	Vi Commands	Options
appreviate	**^E**†	**edcompatible**
map	**^Y**†	**mesg**
unmap		**remap**
stop		**tags**‡

†Simultaneously press the control key and the character key.

‡The **tag** command is present in version 2.13, although the **tags** option is not. This means that, if tags are used with the version 2.13 editor, they are read from a prescribed set of files. You cannot specify alternate names for tag files.

Some size limitations differ between versions 2.13 and 3.5. Most significantly, version 3.5 can accomodate larger files, up to 250,000 lines, as opposed to about 250,000 characters in version 2.13. For details on different limitations, refer to the online lists of changes mentioned above.

Various other features of the editor are noted in the manual as "not available on all v2 editors." This message relates to variants of *ex* designed for computers other than the PDP 11/70; the restrictions generally do not apply to customers of CFO UNIX systems.

Computing Services
September 1981

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 *visual* 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.

Appendix: List of Changes from Version 3.5 to Version 3.6 of the Text Editor ex/vi

- A kernel problem on the version 7 PDP-11 overlay systems which causes bad EMT traps to happen randomly, core dumping the editor, has been programmed around by catching EMT traps.
- A bug which prevented using a screen larger than 48 lines has been fixed.
- A bug which allowed you to set window to a value larger than your screen size has been fixed.
- The screen size limit on non-VM UNIX systems has been increased to 66 lines or 5000 characters, to allow the Ann Arbor Ambassador terminal to be used.
- A bug which caused hangups to be ignored on USG systems has been fixed.
- A bug which caused maps with multiple changes on multiple lines to mess up has been fixed.
- If you get I/O errors, the file is considered "not edited" so that you don't accidently clobber the good file with a munged up buffer.
- An inefficiency in 3.5 which caused the editor to always call **ttyname** has been fixed.
- A bug which prevented the source (**.so**) command from working in an EXINIT or from visual has been fixed.
- A bug which caused readonly to be cleared when reading from a writable file with **r** has been fixed.
- The name **suspend** has been made an alias for **stop**.
- The **stop** command now once again works correctly from command mode.
- On a dumb terminal at 1200 baud, **slowopen** is now the default.
- A bug in the shell script **makeoptions** which searched for a string that appeared earlier in a comment has been fixed.
- A bug that caused an infinite loop when you did **:s/</&/g** has been fixed.
- A bug that caused & with no previous substitution to give "re internal error" has been fixed.
- A bug in the binary search algorithm for **tags** which sometimes prevented the last tag in the file from being found has been fixed.
- Error messages from expreserve no longer output a linefeed, messing up the screen.

- The message from expreserve telling you a buffer was saved when your phone was hung up has be amended to say the "editor was terminated," since a **kill** can also produce that message.

- The **directory** option, which has been broken for over a year, has been fixed.

- The **r** command no longer invokes input mode macros.

- A bug which caused strangeness if you set **wrapmargin** to 1 and typed a line containing a backslash in column 80 has been fixed.

- A bug which caused the **r**<RETURN> at the wrapmargin column to mess up has been fixed.

- On terminals with both scroll reverse and insert line, the least expensive of the two will be used to scroll up. This is usually scroll reverse, which is much less annoying than insert line on terminals such as the mime I and mime 2a.

- A bug which caused vi to estimate the cost of cursor motion without taking into account padding has been fixed.

- The failure of the editor to check counts on ^F and ^B commands has been fixed.

- The **remap** option failed completely if it was turned off. This has been fixed.

- A check of the wrong limit on a buffer for the right hand side of substitutions has been fixed. Overflowing this buffer could produce a core dump.

- A bug causing the editor to go into insert mode if you pressed the RETURN key during an **R** command has been fixed.

- A bug preventing the **+** command from working when you edit a new file has been fixed by making it no longer an error to edit a new file (when you first enter the editor). Instead you are told it is a new file.

- If an error happens when you are writing out a file, such as an interrupt, you are warned that the file is incomplete.

An Introduction to Display Editing with Vi

William Joy

Revised for Versions 3.5 (VAX UNIX) and 2.13 (PDP UNIX) 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 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.

Computer Science
September 1980

Computing Services Notes
See reverse side for "Computing Services Notes" about the new versions of the *vi* editor: versions 3.5 (VAX UNIX) and 2.13 (PDP UNIX). Also see the Appendix for a "List of Changes from Version 3.5 to Version 3.6 of the Text Editor ex/vi."

Computing Services Notes

This manual documents both versions 2.13 and 3.5 of the text editor. Version 2.13 is currently supported on the PDP UNIX Systems operated by Computer Facilities and Operations (CFO); version 3.6 is currently supported on the CFO VAX UNIX system. For VAX UNIX users, the changes from version 3.5 to 3.6 are listed in an appendix to this manual. Also, a cumulative list of changes to the editor from version to version is maintained online on all UNIX systems; to retrieve the information type **help ex news** on PDP UNIX and cat **/usr/news/ex** on VAX UNIX.

A small number of features available with version 3.5 are not found in version 2.13. In most cases, the Manual documents these differences with footnotes stating "Version 3 only." But there are still some instances in which differences are not noted, or are noted ambiguously. Here is a complete list of the editor's commands and options available in version 3.5 but not in 2.13:

Ex Commands	Vi Commands	Options
appreviate	**^E**†	**edcompatible**
map	**^Y**†	**mesg**
unmap		**remap**
stop		**tags**‡

†Simultaneously press the control key and the character key.

‡The **tag** command is present in version 2.13, although the **tags** option is not. This means that, if tags are used with the Version 2.13 editor, they are read from a prescribed set of files. You cannot specify alternate names for tag files.

Some size limitations differ between versions 2.13 and 3.5. Most significantly, version 3.5 can accomodate larger files, up to 250,000 lines, as opposed to about 250,000 characters in version 2.13. For details on different limitations, refer to the online lists of changes mentioned above.

Various other features of the editor are noted in the manual as "not available on all v2 editors." This message relates to variants of *vi* designed for computers other than the PDP 11/70; the restrictions generally do not apply to customers of CFO UNIX systems.

Computing Services
September 1981

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 **r***c*, where *c* is replaced by the correct character. Finally if the character which is incorrect should be replaced by more than one character, give the command **s** which substitutes a string of characters, ending with ESC, for it. If there are a small number of characters which are wrong you can precede **s** with a 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 **df***x* for some *x* now and notice that the *x* character is deleted. Undo this with **u** and then try **dt***x*; the **t** here stands for to, i.e. delete up to the next *x*, but not the *x*. The command **T** is the reverse of **t**.

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

Your file may have tab (^I) characters in it. These characters are represented as a number of spaces expanding to a tab stop, where tab stops are every 8 positions.* 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
f*x*	find *x* forward in line
p	put text back, after cursor or below current line
y	yank operator, for copies and moves
t*x*	up to *x* forward, for operators
F*x*	f backward in line
P	put text back, before cursor or above current line
T*x*	t backward in line

5. High level commands

5.1. Writing, quitting, editing new files

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

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

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

5.2. Escaping to a shell

You can get to a shell to execute a single command by giving a *vi* command of the form **:!***cmd*CR. The system will run the single command *cmd* and when the command finishes, the editor will ask you to hit a RETURN to continue. When you have finished looking at the output on the screen, you should hit RETURN and the editor will clear the screen and redraw it. You can then continue editing. You can also give another **:** command when it asks you for a RETURN; in this case the screen will not be redrawn.

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

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

5.3. Marking and returning

The command '' 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 **:set**CR, or the value of a single option by the command **:set** *opt*?CR. A list of all possible options and their values is generated by **:set all**CR. Set can be abbreviated se. Multiple options can be placed on one line, e.g. **:se ai aw nu**CR.

Options set by the set command only last while you stay in the editor. It is common to want to have certain options set whenever you use the editor. This can be accomplished by creating a list of *ex* commands† 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=10**CR. This causes all lines to be broken at a space at least 10 columns from the right hand edge of the screen.*

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

6.6. Features for editing programs

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

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

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

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

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

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

---

* 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 **:wq**CR. 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 ^VBBBB

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

7. Word Abbreviations ‡‡

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

:ab eecs Electrical Engineering and Computer Sciences

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

7.1. Abbreviations

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

8. Nitty-gritty details

8.1. Line representation in the display

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

The editor only puts full lines on the display; if there is not enough room on the display to fit a logical line, the editor leaves the physical line empty, placing only an @ on the line as a

‡‡ 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 **5**RETURN 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 icCR. The command :se noicCR turns this off.

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

set nomagic

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

↑	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 **:stop**CR. Otherwise, unused.

^[(ESC) Cancels a partially formed command, such as a **z** when no following character has yet been given; terminates inputs on the last line (read by commands such as **:** **/** and **?**); ends insertions of new text into the buffer. If an ESC is given when quiescent in command state, the editor rings the bell or flashes the screen. You can thus hit ESC if you don't know what is happening till the editor rings the bell. If you don't know if you are in insert mode you can type ESC**a**, and then material to be input; the material will be inserted correctly whether or not you were in insert mode when you started (1.5, 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 3c) 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)

Appendix: List of Changes from Version 3.5 to Version 3.6 of the Text Editor ex/vi

- A kernel problem on the version 7 PDP-11 overlay systems which causes bad EMT traps to happen randomly, core dumping the editor, has been programmed around by catching EMT traps.
- A bug which prevented using a screen larger than 48 lines has been fixed.
- A bug which allowed you to set window to a value larger than your screen size has been fixed.
- The screen size limit on non-VM UNIX systems has been increased to 66 lines or 5000 characters, to allow the Ann Arbor Ambassador terminal to be used.
- A bug which caused hangups to be ignored on USG systems has been fixed.
- A bug which caused maps with multiple changes on multiple lines to mess up has been fixed.
- If you get I/O errors, the file is considered "not edited" so that you don't accidently clobber the good file with a munged up buffer.
- An inefficiency in 3.5 which caused the editor to always call **ttyname** has been fixed.
- A bug which prevented the source (**.so**) command from working in an EXINIT or from visual has been fixed.
- A bug which caused readonly to be cleared when reading from a writable file with **r** has been fixed.
- The name **suspend** has been made an alias for **stop**.
- The **stop** command now once again works correctly from command mode.
- On a dumb terminal at 1200 baud, **slowopen** is now the default.
- A bug in the shell script **makeoptions** which searched for a string that appeared earlier in a comment has been fixed.
- A bug that caused an infinite loop when you did **:s/</&/g** has been fixed.
- A bug that caused & with no previous substitution to give "re internal error" has been fixed.
- A bug in the binary search algorithm for **tags** which sometimes prevented the last tag in the file from being found has been fixed.
- Error messages from expreserve no longer output a linefeed, messing up the screen.

- The message from expreserve telling you a buffer was saved when your phone was hung up has be amended to say the "editor was terminated," since a **kill** can also produce that message.

- The **directory** option, which has been broken for over a year, has been fixed.

- The **r** command no longer invokes input mode macros.

- A bug which caused strangeness if you set **wrapmargin** to 1 and typed a line containing a backslash in column 80 has been fixed.

- A bug which caused the **r<RETURN>** at the wrapmargin column to mess up has been fixed.

- On terminals with both scroll reverse and insert line, the least expensive of the two will be used to scroll up. This is usually scroll reverse, which is much less annoying than insert line on terminals such as the mime I and mime 2a.

- A bug which caused vi to estimate the cost of cursor motion without taking into account padding has been fixed.

- The failure of the editor to check counts on ^F and ^B commands has been fixed.

- The **remap** option failed completely if it was turned off. This has been fixed.

- A check of the wrong limit on a buffer for the right hand side of substitutions has been fixed. Overflowing this buffer could produce a core dump.

- A bug causing the editor to go into insert mode if you pressed the RETURN key during an **R** command has been fixed.

- A bug preventing the **+** command from working when you edit a new file has been fixed by making it no longer an error to edit a new file (when you first enter the editor). Instead you are told it is a new file.

- If an error happens when you are writing out a file, such as an interrupt, you are warned that the file is incomplete.

Ex/Edit Command Summary

Computing Services
University of California
Library, 218 Evans Hall
Berkeley, California 94720
415-642-5205

UNX 3.4.1 August 1980

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 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 used without a *name*, prints the name of the current file. If followed by a *name*, renames the current file to *name*.	**:f** "ch10" line 3 of 3 ... **:f ch9** "ch9" [Not edited] line 3 ... :
(1,S)**global** (1,S)**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 "S", and tabs printed as "^I".	**:9l** This is line nineS :

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 on system storage space 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 complete 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,$)write *file* (1,$)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 **:** 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.	

Source Code Control System

User's Guide

L. E. Bonanni
C. A. Salemi

Bell Telephone Laboratories, Incorporated

Source Code Control System
User's Guide

LIST OF FIGURES

Source Code Control System User's Guide

L. E. Bonanni

Bell Laboratories
Piscataway, New Jersey 08854

C. A. Salemi

Bell Laboratories
Piscataway, New Jersey 08854

1. INTRODUCTION

The Source Code Control System (SCCS) is a collection of PWB commands that help individuals or projects control and account for changes to files of text (typically, the source code and documentation of software systems). It is convenient to conceive of SCCS as a custodian of files; it allows retrieval of particular versions of the files, administers changes to them, controls updating privileges to them, and records who made each change, when and where it was made, and why. This is important in environments in which programs and documentation undergo frequent changes (because of maintenance and/or enhancement work), inasmuch as it is sometimes desirable to regenerate the version of a program or document as it was before changes were applied to it. Obviously, this could be done by keeping copies (on paper or other media), but this quickly becomes unmanageable and wasteful as the number of programs and documents increases. SCCS provides an attractive solution because it stores on disk the original file and, whenever changes are made to it, stores only the *changes:* each set of changes is called a "delta."

This document, together with relevant portions of [1], is a complete user's guide to SCCS. This manual contains the following sections:

- *SCCS for Beginners:* How to make an SCCS file, how to update it, and how to retrieve a version thereof.
- *How Deltas Are Numbered:* How versions of SCCS files are numbered and named.
- *SCCS Command Conventions:* Conventions and rules generally applicable to all SCCS commands.
- *SCCS Commands:* Explanation of all SCCS commands, with discussions of the more useful arguments.
- *SCCS Files:* Protection, format, and auditing of SCCS files, including a discussion of the differences between using SCCS as an individual and using it as a member of a group or project. The role of a "project SCCS administrator" is introduced.

2. SCCS FOR BEGINNERS

It is assumed that the reader knows how to log onto a PWB system, create files, and use the text editor [1]. A number of terminal-session fragments are presented below. All of them should be tried: the best way to learn SCCS is to use it.

To supplement the material in this manual, the detailed SCCS command descriptions (appearing in [1]) should be consulted. Section 5 below contains a list of all the SCCS commands. For the time being, however, only basic concepts will be discussed.

2.1 Terminology

Each SCCS file is composed of one or more sets of changes applied to the null (empty) version of the file, with each set of changes usually depending on all previous sets. Each set of changes is called a "delta" and is assigned a name, called the *SCCS IDentification* string (SID), composed of at most four components, only the first two of which will concern us for now; these are the "release" and "level" numbers, separated by a period. Hence, the first delta is called "1.1", the second "1.2", the third "1.3", etc. The release number can also be changed allowing, for example, deltas "2.1", "3.19", etc. The change in the release number usually

indicates a major change to the file.

Each delta of an SCCS file defines a particular version of the file. For example, delta 1.5 defines version 1.5 of the SCCS file, obtained by applying to the null (empty) version of the file the changes that constitute deltas 1.1, 1.2, etc., up to and including delta 1.5 itself, in that order.

2.2 Creating an SCCS File—The "admin" Command

Consider, for example, a file called "lang" that contains a list of programming languages:

```
c
pl/i
fortran
cobol
algol
```

We wish to give custody of this file to SCCS. The following *admin* command (which is used to *administer* SCCS files) creates an SCCS file and initializes delta 1.1 from the file "lang":

```
admin -ilang s.lang
```

All SCCS files *must* have names that begin with "s.", hence, "s.lang". The -i keyletter, together with its value "lang", indicates that *admin* is to create a new SCCS file and *initialize* it with the contents of the file "lang". This initial version is a set of changes applied to the null SCCS file; it is delta 1.1.

The *admin* command replies:

```
No id keywords (cm7)
```

This is a warning message (which may also be issued by other SCCS commands) that is to be ignored for the purposes of this section. Its significance is described in Section 5.1 below. In the following examples, this warning message is not shown, although it may actually be issued by the various command.

The file "lang" should be removed (because it can be easily reconstructed by using the *get* command, below):

```
rm lang
```

2.3 Retrieving a File—The "get" Command

The command:

```
get s.lang
```

causes the creation (retrieval) of the latest version of file "s.lang", and prints the following messages:

```
1.1
5 lines
```

This means that *get* retrieved version 1.1 of the file, which is made up of 5 lines of text. The retrieved text is placed in a file whose name is formed by deleting the "s." prefix from the name of the SCCS file; hence, the file "lang" is created.

The above *get* command simply creates the file "lang" read-only, and keeps no information whatsoever regarding its creation. On the other hand, in order to be able to subsequently apply changes to an SCCS file with the *delta* command (see below), the *get* command must be informed of your intention to do so. This is done as follows:

```
get -e s.lang
```

The -e keyletter causes *get* to create a file "lang" for both reading and writing (so that it may be edited) and places certain information about the SCCS file in another new file, called the *p-file*, that will be read by the *delta* command. The *get* command prints the same messages as before, except that the SID of the version to be created through the use of *delta* is also issued. For example:

```
get -e s.lang
1.1
new delta 1.2
5 lines
```

The file "lang" may now be changed, for example, by:

```
ed lang
27
$a
snobol
ratfor
.
w
41
q
```

2.4 Recording Changes—The "delta" Command

In order to record within the SCCS file the changes that have been applied to "lang", execute:

```
delta s.lang
```

Delta prompts with:

```
comments?
```

the response to which should be a description of why the changes were made; for example:

```
comments? added more languages
```

Delta then reads the *p-file*, and determines what changes were made to the file "lang". It does this by doing its own *get* to retrieve the original version, and by applying *diff*(1)[1] to the original version and the edited version.

---

1. All references of the form *name*(*N*) refer to item *name* in command writeup section *N* of [1].

When this process is complete, at which point the changes to "lang" have been stored in "s.lang", *delta* outputs:

```
1.2
2 inserted
0 deleted
5 unchanged
```

The number "1.2" is the name of the delta just created, and the next three lines of output refer to the number of lines in the file "s.lang".

2.5 More about the "get" Command

As we have seen:

```
get s.lang
```

retrieves the latest version (now 1.2) of the file "s.lang". This is done by starting with the original version of the file and successively applying deltas (the changes) in order, until all have been applied.

For our example, the following commands are all equivalent:

```
get s.lang
get -r1 s.lang
get -r1.2 s.lang
```

The numbers following the -r keyletter are SIDs (see Section 2.1 above). Note that omitting the level number of the SID (as in the second example above) is equivalent to specifying the *highest* level number that exists within the specified release. Thus, the second command requests the retrieval of the latest version in release 1, namely 1.2. The third command specifically requests the retrieval of a particular version, in this case, also 1.2.

Whenever a truly major change is made to a file, the significance of that change is usually indicated by changing the *release* number (first component of the SID) of the delta being made. Since normal, automatic, numbering of deltas proceeds by incrementing the level number (second component of the SID), we must indicate to SCCS that we wish to change the release number. This is done with the *get* command:

```
get -e -r2 s.lang
```

Because release 2 does not exist, *get* retrieves the latest version *before* release 2; it also interprets this as a request to change the release number of the delta we wish to create to 2, thereby causing it to be named 2.1, rather than 1.3. This information is conveyed to *delta* via the *p-file*. *Get* then outputs:

```
1.2
new delta 2.1
7 lines
```

which indicates that version 1.2 has been retrieved and that 2.1 is the version *delta* will create. If the file is now edited, for example, by:

```
ed lang
41
/cobol/d
w
35
q
```

and *delta* executed:

```
delta s.lang
comments? deleted cobol from list of languages
```

we will see, by *delta's* output, that version 2.1 is indeed created:

```
2.1
0 inserted
1 deleted
6 unchanged
```

Deltas may now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release may be created in a similar manner. This process may be continued as desired.

2.6 The "help" Command

If the command:

```
get abc
```

is executed, the following message will be output:

```
ERROR [abc]: not an SCCS file (co1)
```

The string "co1" is a code for the diagnostic message, and may be used to obtain a fuller explanation of that message by use of the *help* command:

```
help co1
```

This produces the following output:

```
co1:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

Thus, *help* is a useful command to use whenever there is any doubt about the meaning of an SCCS message. Fuller explanations of almost all SCCS messages may be found in this manner.

3. HOW DELTAS ARE NUMBERED

It is convenient to conceive of the deltas applied to an SCCS file as the nodes of a tree, in which the root is the initial version of the file. The root delta (node) is normally named "1.1" and successor deltas (nodes) are named "1.2", "1.3", etc. The components of the names of the deltas are called the "release" and the "level" numbers, respectively. Thus, normal naming of successor deltas proceeds by incrementing the level number, which is performed automatically by SCCS whenever a delta is made. In addition, the user may wish to change the *release* number when making a delta, to indicate that a major change is being made. When this is done, the release number also applies to all successor deltas, unless specifically changed again. Thus, the evolution of a particular file may be represented as in Figure 1.

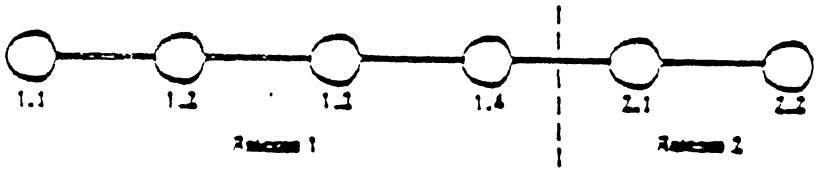


Figure 1. Evolution of an SCCS File

Such a structure may be termed the "trunk" of the SCCS tree. It represents the normal *sequential* development of an SCCS file, in which changes that are part of any given delta are dependent upon *all* the preceding deltas.

However, there are situations in which it is necessary to cause a *branching* in the tree, in that changes applied as part of a given delta are *not* dependent upon all previous deltas. As an example, consider a program which is in production use at version 1.3, and for which development work on release 2 is already in progress. Thus, release 2 may already have some deltas, precisely as shown in Figure 1. Assume that a production user reports a problem in version 1.3, and that the nature of the problem is such that it cannot wait to be repaired in release 2. The changes necessary to repair the trouble will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user, but will *not* affect the changes being applied for release 2 (i.e., deltas 1.4, 2.1, 2.2, etc.).

The new delta is a node on a "branch" of the tree, and its name consists of *four* components, namely, the release and level numbers, as with trunk deltas, plus the "branch" and "sequence" numbers, as follows:

release.level.branch.sequence

The *branch* number is assigned to each branch that is a descendant of a particular trunk delta, with the first such branch being 1, the next one 2, and so on. The *sequence* number is assigned, in order, to each delta on a *particular branch*. Thus, 1.3.1.2 identifies the second delta of the first branch that derives from delta 1.3. This is shown in Figure 2.

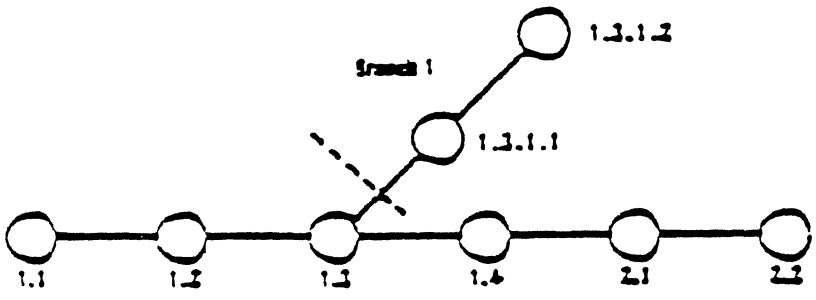


Figure 2. Tree Structure with Branch Deltas

The concept of branching may be extended to any delta in the tree; the naming of the resulting deltas proceeds in the manner just illustrated.

Two observations are of importance with regard to naming deltas. First, the names of trunk deltas contain exactly two components, and the names of branch deltas contain exactly four components. Second, the first two components of the name of branch deltas are always those of the ancestral trunk delta, and the branch component is assigned in the order of creation of the branch, independently of its location relative to the trunk delta. Thus, a branch delta may always be identified as such from its name. Although the ancestral trunk delta may be identified from the branch delta's name, it is *not* possible to determine the *entire* path leading

from the trunk delta to the branch delta. For example, if delta 1.3 has one branch emanating from it, all deltas on that branch will be named 1.3.1.*n*. If a delta on this branch then has another branch emanating from *it*, all deltas on the new branch will be named 1.3.2.*n* (see Figure 3). The only information that may be derived from the name of delta 1.3.2.2 is that it is the *chronologically* second delta on the *chronologically* second branch whose *trunk* ancestor is delta 1.3. In particular, it is *not* possible to determine from the name of delta 1.3.2.2 all of the deltas between it and its trunk ancestor (1.3).

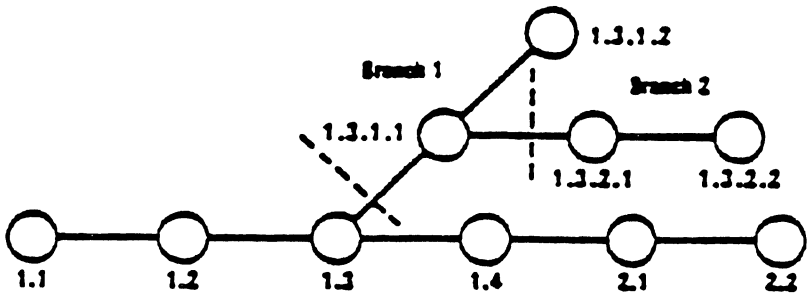


Figure 3. Extending the Branching Concept

It is obvious that the concept of branch deltas allows the generation of arbitrarily complex tree structures. Although this capability has been provided for certain specialized uses, it is strongly recommended that the SCCS tree be kept as simple as possible, because comprehension of its structure becomes extremely difficult as the tree becomes more complex.

4. SCCS COMMAND CONVENTIONS

This section discusses the conventions and rules that apply to SCCS commands. These rules and conventions are generally applicable to *all* SCCS commands, except as indicated below. SCCS commands accept two types of arguments: *keyletter* arguments and *file* arguments.

Keyletter arguments (hereafter called simply "keyletters") begin with a minus sign (—), followed by a lower-case alphabetic character, and, in some cases, followed by a value. These keyletters control the execution of the command to which they are supplied.

File arguments (which may be names of files and/or directories) specify the file(s) that the given SCCS command is to process; naming a directory is equivalent to naming *all* the SCCS files within the directory. Non-SCCS files and unreadable[2] files in the named directories are silently ignored.

In general, file arguments may *not* begin with a minus sign. However, if the name "—" (a lone minus sign) is specified as an argument to a command, the command reads the standard input for lines and takes each line as the *name* of an SCCS file to be processed. The standard input is read until end-of-file. This feature is often used in pipelines [1] with, for example, the *find*(1) or *ls*(1) commands. Again, names of non-SCCS files and of unreadable files are silently ignored.

All keyletters specified for a given command apply to *all* file arguments of that command. All keyletters are processed before any file arguments, with the result that the placement of keyletters is arbitrary (i.e., keyletters may be interspersed with file arguments). File arguments,

2. Because of permission modes (see *chmod*(1)).

however, are processed left to right.

Somewhat different argument conventions apply to the *help*, *what*, *sccsdiff*, and *val* commands (see Sections 5.5, 5.8, 5.9, and 5.11).

Certain actions of various SCCS commands are controlled by *flags* appearing in SCCS files. Some of these flags are discussed below. For a complete description of all such flags, see *admin*(1).

The distinction between the *real user* (see *passwd*(1)) and the *effective user* of a PWB system is of concern in discussing various actions of SCCS commands. For the present, it is assumed that both the real user and the effective user are one and the same (i.e., the user who is logged into a PWB system); this subject is further discussed in Section 6.1.

All SCCS commands that modify an SCCS file do so by writing a temporary copy, called the *x-file*, which ensures that the SCCS file will not be damaged should processing terminate abnormally. The name of the *x-file* is formed by replacing the "s." of the SCCS file name with "x.". When processing is complete, the old SCCS file is removed and the *x-file* is renamed to be the SCCS file. The *x-file* is created in the directory containing the SCCS file, is given the same mode (see *chmod*(1)) as the SCCS file, and is owned by the effective user.

To prevent simultaneous updates to an SCCS file, commands that modify SCCS files create a *lock-file*, called the *z-file*, whose name is formed by replacing the "s." of the SCCS file name with "z.". The *z-file* contains the *process number* [1] of the command that creates it, and its existence is an indication to other commands that that SCCS file is being updated. Thus, other commands that modify SCCS files will not process an SCCS file if the corresponding *z-file* exists. The *z-file* is created with mode 444 (read-only) in the directory containing the SCCS file, and is owned by the effective user. This file exists only for the duration of the execution of the command that creates it. In general, users can ignore *x-files* and *z-files*; they may be useful in the event of system crashes or similar situations.

SCCS commands produce diagnostics (on the diagnostic output [1]) of the form:

ERROR [name-of-file-being-processed]: message text (code)

The *code* in parentheses may be used as an argument to the *help* command (see Section 5.5) to obtain a further explanation of the diagnostic message.

Detection of a fatal error during the processing of a file causes the SCCS command to terminate processing of *that* file and to proceed with the next file, in order, if more than one file has been named.

5. SCCS COMMANDS

This section describes the major features of all the SCCS commands. Detailed descriptions of the commands and of all their arguments are given in the *PWB User's Manual*, and should be consulted for further information. The discussion below covers only the more common arguments of the various SCCS commands.

Because the commands *get* and *delta* are the most frequently used, they are presented first. The other commands follow in approximate order of importance.

The following is a summary of all the SCCS commands and of their major functions:

get	Retrieves versions of SCCS files.
delta	Applies changes (deltas) to the text of SCCS files, i.e., creates new versions.
admin	Creates SCCS files and applies changes to parameters of SCCS files.
prs	Prints portions of an SCCS file in user specified format.
help	Gives explanations of diagnostic messages.
rmdel	Removes a delta from an SCCS file; allows the removal of deltas that were created by mistake.
cdc	Changes the commentary associated with a delta.
what	Searches any PWB file(s) for all occurrences of a special pattern and prints out what follows it; is useful in finding identifying information inserted by the *get* command.
sccsdiff	Shows the differences between any two versions of an SCCS file.
comb	Combines two or more consecutive deltas of an SCCS file into a single delta; often reduces the size of the SCCS file.
val	Validates an SCCS file.

5.1 get

The *get* command creates a text file that contains a particular version of an SCCS file. The particular version is retrieved by beginning with the initial version, and then applying deltas, in order, until the desired version is obtained. The created file is called the *g-file;* its name is formed by removing the "s." from the SCCS file name. The *g-file* is created in the current directory [1] and is owned by the real user. The mode assigned to the *g-file* depends on how the *get* command is invoked, as discussed below.

The most common invocation of *get* is:

```
get s.abc
```

which normally retrieves the latest version on the trunk of the SCCS file tree, and produces (for example) on the standard output [1]:

```
1.3
67 lines
No id keywords (cm7)
```

which indicates that:

1. Version 1.3 of file "s.abc" was retrieved (1.3 is the latest trunk delta).
2. This version has 67 lines of text.
3. No ID keywords were substituted in the file (see Section 5.1.1 for a discussion of ID keywords).

The generated *g-file* (file "abc") is given mode 444 (read-only), since this particular way of invoking *get* is intended to produce *g-files* only for inspection, compilation, etc., and *not* for editing (i.e., *not* for making deltas).

In the case of several file arguments (or directory-name arguments), similar information is given for each file processed, but the SCCS file name precedes it. For example:

```
get s.abc s.def
```

produces:

```
s.abc:
1.3
67 lines
No id keywords (cm7)

s.def:
1.7
35 lines
No id keywords (cm7)
```

5.1.1 ID Keywords

In generating a *g-file* to be used for compilation, it is useful and informative to record the date and time of creation, the version retrieved, the module's name, etc., within the *g-file*, so as to have this information appear in a load module when one is eventually created. SCCS provides a convenient mechanism for doing this automatically. *Identification (ID) keywords* appearing anywhere in the generated file are replaced by appropriate values according to the definitions of these ID keywords. The format of an ID keyword is an upper-case letter enclosed by percent signs (%). For example:

```
%I%
```

is defined as the ID keyword that is replaced by the SID of the retrieved version of a file. Similarly, %H% is defined as the ID keyword for the current date (in the form "mm/dd/yy"), and %M% is defined as the name of the *g-file*. Thus, executing *get* on an SCCS file that contains the PL/I declaration:

```
DCL ID CHAR(100) VAR INIT('%M% %I% %H%');
```

gives (for example) the following:

```
DCL ID CHAR(100) VAR INIT('MODNAME 2.3 07/07/77');
```

When no ID keywords are substituted by *get*, the following message is issued:

```
No id keywords (cm7)
```

This message is normally treated as a warning by *get*, although the presence of the i flag in the SCCS file causes it to be treated as an error (see Section 5.2 for further information).

For a complete list of the approximately twenty ID keywords provided, see *get*(1).

5.1.2 Retrieval of Different Versions

Various keyletters are provided to allow the retrieval of other than the default version of an SCCS file. Normally, the default version is the most recent delta of the highest-numbered release on the *trunk* of the SCCS file tree. However, if the SCCS file being processed has a d (default SID) flag, the SID specified as the value of this flag is used as a default. The default SID is interpreted in exactly the same way as the value supplied with the −r keyletter of *get*.

The −r keyletter is used to specify an SID to be retrieved, in which case the d (default SID) flag (if any) is ignored. For example:

```
get −r1.3 s.abc
```

retrieves version 1.3 of file "s.abc", and produces (for example) on the standard output:

```
1.3
64 lines
```

A branch delta may be retrieved similarly:

```
get -r1.5.2.3 s.abc
```

which produces (for example) on the standard output:

```
1.5.2.3
234 lines
```

When a two- or four-component SID is specified as a value for the -r keyletter (as above) and the particular version does not exist in the SCCS file, an error message results. Omission of the level number, as in:

```
get -r3 s.abc
```

causes retrieval of the *trunk* delta with the highest level number within the given release, if the given release exists. Thus, the above command might output:

```
3.7
213 lines
```

If the given release does not exist, *get* retrieves the *trunk* delta with the highest level number within the highest-numbered existing release that is lower than the given release. For example, assuming release 9 does not exist in file "s.abc", and that release 7 is actually the highest-numbered release below 9, execution of:

```
get -r9 s.abc
```

might produce:

```
7.6
420 lines
```

which indicates that trunk delta 7.6 is the latest version of file "s.abc" below release 9. Similarly, omission of the sequence number, as in:

```
get -r4.3.2 s.abc
```

results in the retrieval of the branch delta with the highest sequence number on the given branch, if it exists. (If the given branch does not exist, an error message results.) This might result in the following output:

```
4.3.2.8
89 lines
```

The –t keyletter is used to retrieve the latest ("top") version in a particular *release* (i.e., when no –r keyletter is supplied, or when its value is simply a release number). The latest version is defined as that delta which was produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5,

```
get –r3 –t s.abc
```

might produce:

```
3.5
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce:

```
3.2.1.5
46 lines
```

5.1.3 Retrieval with Intent to Make a Delta

Specification of the –e keyletter to the *get* command is an indication of the intent to make a delta, and, as such, its use is restricted. The presence of this keyletter causes *get* to check:

1. The *user list* (which is the list of *login* names and/or *group IDs* of users allowed to make deltas (see Section 6.2)) to determine if the login name or group ID of the user executing *get* is on that list. Note that a *null* (empty) user list behaves as if it contained *all* possible login names.
2. That the *release* (R) of the version being retrieved satisfies the relation:

   floor ≤ R ≤ ceiling

   to determine if the release being accessed is a protected release. The *floor* and *ceiling* are specified as *flags* in the SCCS file.
3. That the *release* (R) is not *locked* against editing. The *lock* is specified as a flag in the SCCS file.
4. Whether or not *multiple concurrent edits* are allowed for the SCCS file as specified by the j flag in the SCCS file (multiple concurrent edits are described in Section 5.1.5).

A failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the above checks succeed, the –e keyletter causes the creation of a *g-file* in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user. If a *writable g-file* already exists, *get* terminates with an error. This is to prevent inadvertent destruction of a *g-file* that already exists and is being edited for the purpose of making a delta.

Any ID keywords appearing in the *g-file* are *not* substituted by *get* when the –e keyletter is specified, because the generated *g-file* is to be subsequently used to create another delta, and replacement of ID keywords would cause them to be permanently changed within the SCCS file. In view of this, *get* does not need to check for the presence of ID keywords within the *g-file*, so that the message:

```
No id keywords (cm7)
```

is never output when *get* is invoked with the –e keyletter.

In addition, the –e keyletter causes the creation (or updating) of a *p-file*, which is used to pass information to the *delta* command (see Section 5.1.4).

The following is an example of the use of the −e keyletter:

```
get −e s.abc
```

which produces (for example) on the standard output:

```
1.3
new delta 1.4
67 lines
```

If the −r and/or −t keyletters are used together with the −e keyletter, the version retrieved for editing is as specified by the −r and/or −t keyletters.

The keyletters −i and −x may be used to specify a list (see *get*(1) for the syntax of such a list) of deltas to be *included* and *excluded*, respectively, by *get*. Including a delta means forcing the changes that constitute the particular delta to be included in the retrieved version. This is useful if one wants to apply the same changes to more than one version of the SCCS file. Excluding a delta means forcing it to be *not* applied. This may be used to undo, in the version of the SCCS file to be created, the effects of a previous delta. Whenever deltas are included or excluded, *get* checks for possible interference between such deltas and those deltas that are normally used in retrieving the particular version of the SCCS file. (Two deltas can interfere, for example, when each one changes the same line of the retrieved *g-file*.) Any interference is indicated by a warning that shows the range of lines within the retrieved *g-file* in which the problem may exist. The user is expected to examine the *g-file* to determine whether a problem actually exists, and to take whatever corrective measures (if any) are deemed necessary (e.g., edit the file).

☛ *The* −i *and* −x *keyletters should be used with extreme care.*

The −k keyletter is provided to facilitate regeneration of a *g-file* that may have been accidentally removed or ruined subsequent to the execution of *get* with the −e keyletter, or to simply generate a *g-file* in which the replacement of ID keywords has been suppressed. Thus, a *g-file* generated by the −k keyletter is identical to one produced by *get* executed with the −e keyletter. However, no processing related to the *p-file* takes place.

5.1.4 Concurrent Edits of Different SIDs

The ability to retrieve different versions of an SCCS file allows a number of deltas to be "in progress" at any given time. This means that a number of *get* commands with the −e keyletter may be executed on the same file, provided that no two executions retrieve the same version (unless multiple concurrent edits are allowed, see Section 5.1.5).

The *p-file* (which is created by the *get* command invoked with the −e keyletter) is named by replacing the "s." in the SCCS file name with "p.". It is created in the directory containing the SCCS file, is given mode 644 (readable by everyone, writable only by the owner), and is owned by the effective user. The *p-file* contains the following information for each delta that is still "in progress":[3]

- The SID of the retrieved version.
- The SID that will be given to the new delta when it is created.
- The login name of the real user executing *get*.

The first execution of "get −e" causes the *creation* of the *p-file* for the corresponding SCCS file. Subsequent executions only *update* the *p-file* by inserting a line containing the above information. Before inserting this line, however, *get* checks that no entry already in the *p-file*

3. Other information may be present, but is not of concern here. See *get*(1) for further discussion.

specifies is already retrieved the SID of the version to be retrieved, unless multiple concurrent edits are allowed.

If both checks succeed, the user is informed that other deltas are in progress, and processing continues. If either check fails, an error message results. It is important to note that the various executions of *get* should be carried out from different directories. Otherwise, only the first execution will succeed, since subsequent executions would attempt to over-write a *writable g-file*, which is an SCCS error condition. In practice, such multiple executions are performed by different users,[4] so that this problem does not arise, since each user normally has a different working directory [1].

Table 1 shows, for the most useful cases, what version of an SCCS file is retrieved by *get* as well as the SID of the version to be eventually created by *delta*, as a function of the SID specified to *get*.

5.1.5 Concurrent Edits of the Same SID

Under normal conditions, *gets* for editing (−e keyletter is specified) based on the same SID are not permitted to occur concurrently. That is, *delta* must be executed before a subsequent *get* for editing is executed at the same SID as the previous *get*. However, multiple concurrent edits (defined to be two or more *successive* executions of *get* for editing based on the same retrieved SID) *are* allowed if the j flag is set in the SCCS file. Thus:

```
get −e s.abc
1.1
new delta 1.2
5 lines
```

may be immediately followed by:

```
get −e s.abc
1.1
new delta 1.1.1.1
5 lines
```

without an intervening execution of *delta*. In this case, a *delta* command corresponding to the first *get* produces delta 1.2 (assuming 1.1 is the latest (most recent) trunk delta), and the *delta* command corresponding to the second *get* produces delta 1.1.1.1.

5.1.6 Keyletters That Affect Output

Specification of the −p keyletter causes *get* to write the retrieved text to the standard output, rather than to a *g-file*. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. This may be used, for example, to create *g-files* with arbitrary names:

```
get −p s.abc > arbitrary-filename
```

The −p keyletter is particularly useful when used with the "!" or "$" arguments of the PWB *send*(1) command. For example:

```
send MOD=s.abc REL=3 compile
```

4. See Section 6.1 for a discussion of how different users are permitted to use SCCS commands on the same files.

TABLE 1. Determination of New SID

Case	*SID Specified**	*−b Keyletter Used†*	*Other Conditions*	*SID Retrieved*	*SID of Delta to be Created*
1.	none‡	no	R defaults to mR	mR.mL	mR.(mL + 1)
2.	none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB + 1).1
3.	R	no	R > mR	mR.mL	R.1§
4.	R	no	R = mR	mR.mL	mR.(mL + 1)
5.	R	yes	R > mR	mR.mL	mR.mL.(mB + 1).1
6.	R	yes	R = mR	mR.mL	mR.mL.(mB + 1).1
7.	R	—	R < mR and R does *not* exist	hR.mL**	hR.mL.(mB + 1).1
8.	R	—	Trunk successor in release > R and R exists	R.mL	R.mL.(mB + 1).1
9.	R.L	no	No trunk successor	R.L	R.(L + 1)
10.	R.L	yes	No trunk successor	R.L	R.L.(mB + 1).1
11.	R.L	—	Trunk successor in release ≥ R	R.L	R.L.(mB + 1).1
12.	R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS + 1)
13.	R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB + 1).1
14.	R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S + 1)
15.	R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB + 1).1
16.	R.L.B.S	—	Branch successor	R.L.B.S	R.L.(mB + 1).1

* "R", "L", "B", and "S" are the "release", "level", "branch", and "sequence" components of the SID, respectively; "m" means "maximum". Thus, for example, "R.mL" means "the maximum level number within release R"; "R.L.(mB + 1).1" means "the first sequence number on the *new* branch (i.e., maximum branch number plus 1) of level L within release R". Note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components *must* exist.

† The −b keyletter is effective only if the b flag (see *admin*(1)) is present in the file. In this table, an entry of "−" means "irrelevant".

‡ This case applies if the d (default SID) flag is *not* present in the file. If the d flag *is* present in the file, then the SID obtained from the d flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

§ This case is used to force the creation of the *first* delta in a *new* release.

** "hR" is the highest *existing* release that is lower than the specified, *nonexistent*, release R.

if file "compile" contains:

```
//plicomp job job-card-information
//step1 exec plickc
//pli.sysin dd *
~-s
~!get -p -rREL MOD
/*
//
```

will *send* the highest level of release 3 of file "s.abc". Note that the line "~-s", which causes *send*(1) to make ID keyword substitutions before detecting and interpreting control lines, is necessary if *send*(1) is to substitute "s.abc" for MOD and "3" for REL in the line "~!get -p -rREL MOD".

The −s keyletter suppresses all output that is *normally* directed to the standard output. Thus, the SID of the retrieved version, the number of lines retrieved, etc., are not output. This does not, however, affect messages to the diagnostic output. This keyletter is used to prevent non-diagnostic messages from appearing on the user's terminal, and is often used in conjunction with the −p keyletter to "pipe" the output of *get*, as in:

```
get −p −s s.abc | nroff
```

The −g keyletter is supplied to suppress the actual retrieval of the text of a version of the SCCS file. This may be useful in a number of ways. For example, to verify the existence of a particular SID in an SCCS file, one may execute:

```
get −g −r4.3 s.abc
```

This outputs the given SID if it exists in the SCCS file, or it generates an error message, if it does not. Another use of the −g keyletter is in regenerating a *p-file* that may have been accidentally destroyed:

```
get −e −g s.abc
```

The −l keyletter causes the creation of an *l-file*, which is named by replacing the "s." of the SCCS file name with "l.". This file is created in the current directory, with mode 444 (read-only), and is owned by the real user. It contains a table (whose format is described in *get*(1)) showing which deltas were used in constructing a particular version of the SCCS file. For example:

```
get −r2.3 −l s.abc
```

generates an *l-file* showing which deltas were applied to retrieve version 2.3 of the SCCS file. Specifying a *value* of "p" with the −l keyletter, as in:

```
get −lp −r2.3 s.abc
```

causes the generated output to be written to the standard output rather than to the *l-file*. Note that the −g keyletter may be used with the −l keyletter to suppress the actual retrieval of the text.

The −m keyletter is of use in identifying, line by line, the changes applied to an SCCS file. Specification of this keyletter causes each line of the generated *g-file* to be preceded by the SID of the delta that caused that line to be inserted. The SID is separated from the text of the line by a tab character.

The −n keyletter causes each line of the generated *g-file* to be preceded by the value of the %M% ID keyword (see Section 5.1.1) and a tab character. The −n keyletter is most often used in a pipeline with *grep*(1). For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory, the following may be executed:

```
get −p −n −s directory | grep pattern
```

If both the −m and −n keyletters are specified, each line of the generated *g-file* is preceded by the value of the %M% ID keyword and a tab (this is the effect of the −n keyletter), followed by the line in the format produced by the −m keyletter. Because use of the −m keyletter and/or the −n keyletter causes the contents of the *g-file* to be modified, such a *g-file* must *not* be used for creating a delta. Therefore, neither the −m keyletter nor the −n keyletter may be specified together with the −e keyletter.

See *get*(1) for a full description of additional *get* keyletters.

5.2 delta

The *delta* command is used to incorporate the changes made to a *g-file* into the corresponding SCCS file, i.e., to create a delta, and, therefore, a new version of the file.

Invocation of the *delta* command requires the existence of a *p-file* (see Sections 5.1.3 and 5.1.4). *Delta* examines the *p-file* to verify the presence of an entry containing the user's login name. If none is found, an error message results. *Delta* also performs the same permission checks that *get* performs when invoked with the –e keyletter. If all checks are successful, *delta* determines what has been changed in the *g-file*, by comparing it (via *diff*(1)) with its own, temporary copy of the *g-file* as it was before editing. This temporary copy of the *g-file* is called the *d-file* (its name is formed by replacing the "s." of the SCCS file name with "d.") and is obtained by performing an internal *get* at the SID specified in the *p-file* entry.

The required *p-file* entry is the one containing the login name of the user executing *delta*, because the user who retrieved the *g-file* must be the one who will create the delta. However, if the login name of the user appears in more than one entry (i.e., the same user executed *get* with the –e keyletter more than once on the same SCCS file), the –r keyletter must be used with *delta* to specify an SID that uniquely identifies the *p-file* entry[5]. This entry is the one used to obtain the SID of the delta to be created.

In practice, the most common invocation of *delta* is:

```
delta s.abc
```

which prompts on the standard output (but only if it is a terminal):

```
comments?
```

to which the user replies with a description of why the delta is being made, terminating the reply with a newline character. The user's response may be up to 512 characters long, with newlines *not* intended to terminate the response escaped by "\".

If the SCCS file has a v flag, *delta* first prompts with:

```
MRs?
```

on the standard output. (Again, this prompt is printed only if the standard output is a terminal.) The standard input is then read for MR[6] numbers, separated by blanks and/or tabs, terminated in the same manner as the response to the prompt "comments?".

The –y and/or –m keyletters are used to supply the commentary (comments and MR numbers, respectively) on the command line, rather than through the standard input. For example:

```
delta –y"descriptive comment" –m"mrnum1 mrnum2" s.abc
```

In this case, the corresponding prompts are not printed, and the standard input is not read. The –m keyletter is allowed only if the SCCS file has a v flag. These keyletters are useful when *delta* is executed from within a *Shell procedure* (see *sh*(1)).

The commentary (comments and/or MR numbers), whether solicited by *delta* or supplied via keyletters, is recorded as part of the entry for the delta being created, and applies to *all* SCCS files processed by the same invocation of *delta*. This implies that if *delta* is invoked with more than one file argument, and the first file named has a v flag, all files named must have this flag. Similarly, if the first file named does not have this flag, then none of the files named may have it. Any file that does not conform to these rules is not processed.

---

5. The SID specified may be either the SID retrieved by *get*, or the SID *delta* is to create.
6. In a tightly controlled environment, it is expected that deltas are created only as a result of some trouble report, change request, trouble ticket, etc. (collectively called here Modification Requests, or MRs) and that it is desirable or necessary to record such MR number(s) within each delta.

When processing is complete, *delta* outputs (on the standard output) the SID of the created delta (obtained from the *p-file* entry) and the counts of lines inserted, deleted, and left unchanged by the delta. Thus, a typical output might be:

```
1.4
14 inserted
7 deleted
345 unchanged
```

It is possible that the counts of lines reported as inserted, deleted, or unchanged by *delta* do not agree with the user's perception of the changes applied to the *g-file*. The reason for this is that there usually are a number of ways to describe a set of such changes, especially if lines are moved around in the *g-file*, and *delta* is likely to find a description that differs from the user's perception. However, the *total* number of lines of the new delta (the number inserted plus the number left unchanged) should agree with the number of lines in the edited *g-file*.

If, in the process of making a delta, *delta* finds no ID keywords in the edited *g-file*, the message:

```
No id keywords (cm7)
```

is issued after the prompts for commentary, but before any other output. This indicates that any ID keywords that may have existed in the SCCS file have been replaced by their values, or deleted during the editing process. This could be caused by creating a delta from a *g-file* that was created by a *get* without the **-e** keyletter (recall that ID keywords are replaced by *get* in that case), or by accidentally deleting or changing the ID keywords during the editing of the *g-file*. Another possibility is that the file may never have had any ID keywords. In any case, it is left up to the user to determine what remedial action is necessary, but the delta is made, unless there is an **i** flag in the SCCS file, indicating that this should be treated as a fatal error. In this last case, the delta is not created.

After processing of an SCCS file is complete, the corresponding *p-file* entry is removed from the *p-file*.[7] If there is only *one* entry in the *p-file*, then the *p-file* itself is removed.

In addition, *delta* removes the edited *g-file*, unless the **-n** keyletter is specified. Thus:

```
delta -n s.abc
```

will keep the *g-file* upon completion of processing.

The **-s** ("silent") keyletter suppresses all output that is normally directed to the standard output, other than the prompts "comments?" and "MRs?". Thus, use of the **-s** keyletter together with the **-y** keyletter (and possibly, the **-m** keyletter) causes *delta* neither to read the standard input nor to write the standard output.

The differences between the *g-file* and the *d-file* (see above), which constitute the delta, may be printed on the standard output by using the **-p** keyletter. The format of this output is similar to that produced by *diff*(1).

5.3 admin

The *admin* command is used to *administer* SCCS files, that is, to create new SCCS files and to change parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of keyletters or are assigned default values if no keyletters are supplied. The same keyletters are used to change the parameters of existing files.

---

7. All updates to the *p-file* are made to a temporary copy, the *q-file*, whose use is similar to the use of the *x-file*, which is described in Section 4 above.

Two keyletters are supplied for use in conjunction with detecting and correcting "corrupted" SCCS files, and are discussed in Section 6.3 below.

Newly-created SCCS files are given mode 444 (read-only) and are owned by the effective user.

Only a user with write permission in the directory containing the SCCS file may use the *admin* command upon that file.

5.3.1 Creation of SCCS Files

An SCCS file may be created by executing the command:

```
admin -ifirst s.abc
```

in which the value ("first") of the -i keyletter specifies the name of a file from which the text of the *initial* delta of the SCCS file "s.abc" is to be taken. Omission of the value of the -i keyletter indicates that *admin* is to read the standard input for the text of the initial delta. Thus, the command:

```
admin -i s.abc < first
```

is equivalent to the previous example. If the text of the initial delta does not contain ID keywords, the message:

```
No id keywords (cm7)
```

is issued by *admin* as a warning. However, if the same invocation of the command also sets the i flag (not to be confused with the -i keyletter), the message is treated as an error and the SCCS file is not created. Only *one* SCCS file may be created at a time using the -i keyletter.

When an SCCS file is created, the *release* number assigned to its first delta is normally "1", and its *level* number is always "1". Thus, the first delta of an SCCS file is normally "1.1". The -r keyletter is used to specify the release number to be assigned to the first delta. Thus:

```
admin -ifirst -r3 s.abc
```

indicates that the first delta should be named "3.1" rather than "1.1". Because this keyletter is only meaningful in creating the first delta, its use is only permitted with the -i keyletter.

5.3.2 Inserting Commentary for the Initial Delta

When an SCCS file is created, the user may choose to supply commentary stating the reason for creation of the file. This is done by supplying comments (-y keyletter) and/or MR numbers[8] (-m keyletter) in exactly the same manner as for *delta*. If comments (-y keyletter) are omitted, a comment line of the form:

```
date and time created YY/MM/DD HH:MM:SS by logname
```

is automatically generated.

If it is desired to supply MR numbers (-m keyletter), the v flag must also be set (using the -f keyletter described below). The v flag simply determines whether or not MR numbers must be supplied when using any SCCS command that modifies a *delta commentary* (see *sccsfile*(5)) in the SCCS file. Thus:

```
admin -ifirst -mmrnum1 -fv s.abc
```

Note that the -y and -m keyletters are only effective if a new SCCS file is being created.

---

8. The creation of an SCCS file may sometimes be the direct result of an MR.

5.3.3 Initialization and Modification of SCCS File Parameters

The portion of the SCCS file reserved for *descriptive text* (see Section 6.2) may be initialized or changed through the use of the –t keyletter. The descriptive text is intended as a summary of the contents and purpose of the SCCS file, although its contents may be arbitrary, and it may be arbitrarily long.

When an SCCS file is being created and the –t keyletter is supplied, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the command:

```
admin –ifirst –tdesc s.abc
```

specifies that the descriptive text is to be taken from file "desc".

When processing an *existing* SCCS file, the –t keyletter specifies that the descriptive text (if any) currently in the file is to be *replaced* with the text in the named file. Thus:

```
admin –tdesc s.abc
```

specifies that the descriptive text of the SCCS file is to be replaced by the contents of "desc"; omission of the file name after the –t keyletter as in:

```
admin –t s.abc
```

causes the *removal* of the descriptive text from the SCCS file.

The *flags* (see Section 6.2) of an SCCS file may be initialized and changed, or deleted through the use of the –f and –d keyletters, respectively. The flags of an SCCS file are used to direct certain actions of the various commands. See *admin*(1) for a description of all the flags. For example, the l flag specifies that the warning message stating there are no ID keywords contained in the SCCS file should be treated as an error, and the d (default SID) flag specifies the default version of the SCCS file to be retrieved by the *get* command. The –f keyletter is used to set a flag and, possibly, to set its value. For example:

```
admin –ifirst –fl –fmmodname s.abc
```

sets the l flag and the m (module name) flag. The value "modname" specified for the m flag is the value that the *get* command will use to replace the %M% ID keyword. (In the absence of the m flag, the name of the *g-file* is used as the replacement for the %M% ID keyword.) Note that several –f keyletters may be supplied on a single invocation of *admin*, and that –f keyletters may be supplied whether the command is creating a new SCCS file or processing an existing one.

The –d keyletter is used to delete a flag from an SCCS file, and may only be specified when processing an existing file. As an example, the command:

```
admin –dm s.abc
```

removes the m flag from the SCCS file. Several –d keyletters may be supplied on a single invocation of *admin*, and may be intermixed with –f keyletters.

SCCS files contain a list (*user list*) of login names and/or group IDs of users who are allowed to create deltas (see Sections 5.1.3 and 6.2). This list is empty by default, which implies that *anyone* may create deltas. To add login names and/or group IDs to the list, the –a keyletter is used. For example:

```
admin –axyz –awql –a1234 s.abc
```

adds the login names "xyz" and "wql" and the group ID "1234" to the list. The –a keyletter may be used whether *admin* is creating a new SCCS file or processing an existing one, and may appear several times. The –e keyletter is used in an analogous manner if one wishes to remove ("erase") login names or group IDs from the list.

5.4 prs

Prs is used to print on the standard output all or parts of an SCCS file (see Section 6.2) in a format, called the output *data specification*, supplied by the user via the —d keyletter. The data specification is a string consisting of SCCS file *data keywords*[9] interspersed with optional user text.

Data keywords are replaced by appropriate values according to their definitions. For example:

```
:I:
```

is defined as the data keyword that is replaced by the SID of a specified delta. Similarly, :F: is defined as the data keyword for the SCCS file name currently being processed, and :C: is defined as the comment line associated with a specified delta. All parts of an SCCS file have an associated data keyword. For a complete list of the data keywords, see *prs*(1).

There is no limit to the number of times a data keyword may appear in a data specification. Thus, for example:

```
prs —d":I: this is the top delta for :F: :I:" s.abc
```

may produce on the standard output:

```
2.1 this is the top delta for s.abc 2.1
```

Information may be obtained from a single delta by specifying the SID of that delta using the —r keyletter. For example:

```
prs —d":F:: :I: comment line is: :C:" —r1.4 s.abc
```

may produce the following output:

```
s.abc: 1.4 comment line is: THIS IS A COMMENT
```

If the —r keyletter is *not* specified, the value of the SID defaults to the most recently created delta.

In addition, information from a *range* of deltas may be obtained by specifying the —l or —e keyletters. The —e keyletter substitutes data keywords for the SID designated via the —r keyletter and all deltas created *earlier*. The —l keyletter substitutes data keywords for the SID designated via the —r keyletter and all deltas created *later*. Thus, the command:

```
prs —d:I: —r1.4 —e s.abc
```

may output:

```
1.4
1.3
1.2.1.1
1.2
1.1
```

---

9 Not to be confused with *get ID keywords*.

and the command:

```
prs -d:I: -r1.4 -i s.abc
```

may produce:

```
3.3
3.2
3.1
2.2.1.1
2.2
2.1
1.4
```

Substitution of data keywords for *all* deltas of the SCCS file may be obtained by specifying both the -e and -i keyletters.

5.5 help

The *help* command prints explanations of SCCS commands and of messages that these commands may print. Arguments to *help*, zero or more of which may be supplied, are simply the names of SCCS commands or the code numbers that appear in parentheses after SCCS messages. If no argument is given, *help* prompts for one. *Help* has no concept of *keyletter* arguments or *file* arguments. Explanatory information related to an argument, if it exists, is printed on the standard output. If no information is found, an error message is printed. Note that each argument is processed independently, and an error resulting from one argument will *not* terminate the processing of the other arguments.

Explanatory information related to a command is a synopsis of the command. For example:

```
help ge5 rmdel
```

produces:

```
ge5:
"nonexistent sid"
The specified sid does not exist in the
given file.
Check for typos.

rmdel:
   rmdel -rSID name ...
```

5.6 rmdel

The *rmdel* command is provided to allow *removal* of a delta from an SCCS file, though its use should be reserved for those cases in which incorrect, global changes were made a part of the delta to be removed.

The delta to be removed must be a "leaf" delta. That is, it must be the latest (most recently created) delta on its branch or on the trunk of the SCCS file tree. In Figure 3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed; once they are removed, then deltas 1.3.2.1 and 2.1 can be removed, and so on.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must either be the one who created the delta being removed, or be the owner of the SCCS file and its directory.

The –r keyletter, which is mandatory, is used to specify the *complete* SID of the delta to be removed (i.e., it must have two components for a trunk delta, and four components for a branch delta). Thus:

```
rmdel –r2.3 s.abc
```

specifies the removal of (trunk) delta "2.3" of the SCCS file. Before removal of the delta, *rmdel* checks that the *release* number (R) of the given SID satisfies the relation:

floor ≤ R ≤ ceiling

Rmdel also checks that the SID specified is *not* that of a version for which a *get* for editing has been executed and whose associated *delta* has not yet been made. In addition, the login name or group ID of the user must appear in the file's *user list*, or the *user list* must be empty. Also, the release specified can not be *locked* against editing (i.e., if the l flag is set (see *admin*(1)), the release specified *must* not be contained in the list). If these conditions are not satisfied, processing is terminated, and the delta is not removed. After the specified delta has been removed, its type indicator in the *delta table* of the SCCS file (see Section 6.2) is changed from "D" (for "delta") to "R" (for "removed").

5.7 cdc

The *cdc* command is used to *change* a delta's commentary that was supplied when that delta was created. Its invocation is analogous to that of the *rmdel* command, except that the delta to be processed is *not* required to be a leaf delta. For example:

```
cdc –r3.4 s.abc
```

specifies that the commentary of delta "3.4" of the SCCS file is to be changed.

The *new* commentary is solicited by *cdc* in the same manner as that of *delta*. The old commentary associated with the specified delta is kept, but it is preceded by a comment line indicating that it has been changed (i.e., superseded), and the new commentary is entered ahead of this comment line. The "inserted" comment line records the login name of the user executing *cdc* and the time of its execution.

Cdc also allows for the deletion of selected MR numbers associated with the specified delta. This is specified by preceding the selected MR numbers by the character "!". Thus:

```
cdc –r1.4 s.abc
MRs? mrnum3 !mrnum1
comments? deleted wrong MR number and inserted correct MR number
```

inserts "mrnum3" and deletes "mrnum1" for delta 1.4.

5.8 what

The *what* command is used to find identifying information within *any* PWB file whose name is given as an argument to *what*. Directory names and a name of "–" (a lone minus sign) are *not* treated specially, as they are by other SCCS commands, and no *keyletters* are accepted by the command.

What searches the given file(s) for all occurrences of the string "@(#)", which is the replacement for the %Z% ID keyword (see *get*(1)), and prints (on the standard output) what follows that string until the first double quote ("), greater than (>), backslash (\), newline, or

(non-printing) NUL character. Thus, for example, if the SCCS file "s.prog.c" (which is a C program), contains the following line (the %M% and %I% ID keywords were defined in Section 5.1.1):

```
char id[] "%Z%%M%:%I%";
```

and then the command:

```
get -r3.4 s.prog.c
```

is executed, and finally the resulting *g-file* is compiled to produce "prog.o" and "a.out", then the command:

```
what prog.c prog.o a.out
```

produces:

```
prog.c:
   prog.c:3.4
prog.o:
   prog.c:3.4
a.out:
   prog.c:3.4
```

The string searched for by *what* need not be inserted via an ID keyword of *get* it may be inserted in any convenient manner.

5.9 scsdiff

The *scsdiff* command determines (and prints on the standard output) the differences between two specified versions of one or more SCCS files. The versions to be compared are specified by using the -r keyletter, whose format is the same as for the *get* command. The two versions *must* be specified as the first two arguments to this command in the order in which they were created, i.e., the older version is specified first. Any following keyletters are interpreted as arguments to the *pr*(1) command (which actually prints the differences) and must appear before any file names. SCCS files to be processed are named last. Directory names and a name of "-" (a lone minus sign) are *not* acceptable to *scsdiff*.

The differences are printed in the form generated by *diff*(1). The following is an example of the invocation of *scsdiff*:

```
scsdiff -r3.4 -r5.6 s.abc
```

5.10 comb

Comb generates a *Shell procedure* (see *sh*(1)) which attempts to reconstruct the named SCCS files so that the reconstructed files are smaller than the originals. The generated Shell procedure is written on the standard output.

Named SCCS files are reconstructed by discarding unwanted deltas and combining specified other deltas. The intended use is for those SCCS files that contain deltas that are so old that they are no longer useful. It is *not* recommended that *comb* be used as a matter of routine; its use should be restricted to a *very* small number of times in the life of an SCCS file.

In the absence of any keyletters, *comb* preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the "shape" of the SCCS file tree. The effect of this is to eliminate "middle" deltas on the trunk and on all branches of the tree. Thus, in Figure 3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated. Some of the keyletters are summarized as follows:

The -p keyletter specifies the oldest delta that is to be preserved in the reconstruction. All older deltas are discarded.

Document Formatting on UNIX†
Using the -ms Macros

Joel Kies

Computing Services
University of California
Berkeley, California 94720

ABSTRACT

This document describes the supported commands used for producing formatted papers, such as dissertations and journal articles, on the UNIX computer system. It is intended to be the main source of information on formatting documents with *nroff* or *troff* and the *-ms macro package*. The reader is assumed to have basic familiarity with UNIX and with a text editor such as *ex*, *edit*, or *vi*.

This paper is based on the Bell Laboratories manual *Typing Documents on the UNIX System: Using the -ms Macros with Troff and Nroff*, by M. E. Lesk, and it replaces that document for U.C. Berkeley UNIX users.

The following people have contributed very substantially to this document by their suggestions and criticism: Ricki Blau, John Kunze, Bob Levinson, Gail Moyer, Betty Nelson, Cindy Nelson, and Ken Wahl.

September 16, 1980

†UNIX is a trademark of Bell Laboratories.

Document Formatting on UNIX
Using the -ms Macros

Contents

Document Formatting on UNIX
Using the -ms Macros

1. Introduction

This document describes a package of commands used in producing formatted papers, such as reports, dissertations, and journal articles, on the UNIX system. The package, called the *-ms macros*, provides commands for paragraphs, section headings, running page titles, footnotes, multi-column format, cover sheets, and other features. To use the facilities described in this paper, you need to have a general familiarity with UNIX and with a text editor such as *ex*, *edit*, or *vi*.

UNIX offers several related formatting programs, suited to handling tables and mathematics as well as ordinary text. A number of separate writeups on these programs are available; an annotated list is included later in this paper (see section 7). This paper should be read first, however, since it provides both a description of the most commonly-used formatting commands and an overview of the related programs.†

The main formatting programs, *nroff* and *troff*, read one or more UNIX files containing both the text to be formatted and commands specifying how the output should look. From this, the programs produce formatted output: nroff for typewriter-like terminals, troff for a phototypesetter. Although they are separate programs, nroff and troff are very compatible: they share the same command language and work in such a way that it is possible for nroff to produce typewriter or line printer output, and troff to produce phototypesetter output, from the same input file. For convenience, we will refer usually to nroff. It should be understood that what is said applies also to troff unless stated otherwise.

1.1. Filling, Adjusting, and Hyphenation

Normally, the text of a document is typed on lines of varying length even though the typist wants lines of uniform length in the finished document. One of nroff's most important functions is *filling*, the process of collecting words from the input file and placing them on an output line until no more will fit within a given line length. *Hyphenation* is also provided, so that a line may be completed with part of a word to obtain a line of the right length. *Adjusting* is performed after the line has been filled: spaces are inserted between words as necessary to bring the text exactly to the right margin.‡ Filling and adjusting are illustrated by the following examples.

Text that is not filled:

The Caterpillar and Alice
looked at each other
for some time in silence:
at last
the Caterpillar took the hookah
out of its mouth,
and addressed her in a sleepy,
languid voice.

Filled but not adjusted:

The Caterpillar and Alice looked at each other for some time in silence: at last the Caterpillar took the hookah out of its mouth, and addressed her in a sleepy, languid voice.

†This document is based on, and replaces for U.C. Berkeley users, the Bell Laboratories manual *Typing Documents on the UNIX System: Using the -ms Macros with Troff and Nroff* by M. E. Lesk (Murray Hill, 1978).

‡It is possible to obtain a ragged right margin, as explained in section 3, below.

Filled and adjusted:

The Caterpillar and Alice looked at each other for some time in silence: at last the Caterpillar took the hookah out of its mouth, and addressed her in a sleepy, languid voice.

Given a file of input consisting only of lines of text (i.e., without any formatting commands), nroff would produce simply a continuous stream of filled, adjusted and hyphenated output. Additional operations such as producing paragraphs, providing margins at tops and bottoms of pages, and saving footnotes to be printed at the bottoms of pages, must be requested by means of formatting commands.

1.2. Purposes of a Macro Package

Nroff provides a flexible, sophisticated command language for requesting operations of the sort just mentioned. Largely because of its high degree of flexibility, however, this language can be very difficult to use. Even a relatively simple formatting task such as beginning a paragraph is a multi-step process in the nroff language. For most documents, it is advantageous to use instead the commands provided by a macro package. A macro is simply a *predefined sequence of nroff commands and/or text* which you can invoke by including just one command in your input file. This makes it possible to handle repetitious tasks, such as starting paragraphs, by typing one command each time instead of several. The -ms package has simple commands for a large number of common formatting tasks.

The macro package has other kinds of functions as well, some of which are less visible but equally important. Nroff, although it provides commands and mechanisms for arranging page layouts with top and bottom margins, page numbers, and running titles automatically placed on every page, doesn't do any of these things on its own; it requires instructions. The -ms macro package supplies such instructions "behind the scenes." It takes care of some of the more difficult programming problems involved in handling footnotes and page transitions, and it sets up a page layout style by default, simply by virtue of your invoking the macro package. It does this in a way, however, that leaves you a great deal of control over the formatting style should you wish to change things.

In general, the commands and other features of -ms are designed to be used instead of the more numerous building-blocks of the nroff language.† The macro package offers a limited subset of the wide range of formatting possibilities afforded by nroff; but it compensates for its limitations by its ease of use.

1.3. Typing an Input File

An input file for nroff, containing text and formatting commands, is created with the text editor. Here is a small example of an input file:

```
.TL
Simple Sample Document
.PP
These are a few lines of sample text.
It doesn't matter whether they are
long or short,
because nroff or troff
will take care of that later.
```

Notice that in this file, some lines contain nothing but text while the others, beginning with a period, contain formatting commands. There are several rules to observe when typing an input file.

First, here are some of the rules with regard to text:

- A line of text should normally end with the end of a word, along with any trailing punctuation. Nroff assumes this will be the case and always inserts a space between whatever ends one line of input text and whatever begins the next. This means, for instance, that you should not break a hyphenated word such as "waistcoat-pocket" between two lines in the input file.
- Lines in the input file should start with characters other than a space. A space at the beginning of an input line causes a *break* at that point in the output—nroff skips immediately to a new output line, interrupting the process of filling and

†There are some exceptions to this rule, discussed in section 3.

adjusting.

- Although text lines can vary in length, it is a good idea to keep them fairly short and type the RETURN key at ends of phrases and ends of sentences. There are two reasons for this. First, it makes the input file easy to modify later. Second, whenever sentence-ending punctuation (period, question mark, or exclamation mark) occurs at the end of an input line, nroff leaves two additional spaces following the sentence. To obtain this effect consistently, always start a new input line when starting a new sentence.

Other rules govern the way commands are typed.

- A period or an apostrophe (') as the first character on a line indicates to nroff that the line contains a formatting command.† It would be an error to type a line of text beginning with either of these *control characters*; nroff would try to interpret the line as a command, and the result at best would be the disappearance of that text from the formatted output.
- Following the control character is the one- or two-character *name* of a formatting command. The names of commands in the –ms macro package usually consist of one or two capital letters; a few consist of one capital letter and one digit. Names of commands in the nroff language consist of lower-case letters, or a lower-case letter and a digit.
- Some commands occur by themselves on a line; others can take one or more additional pieces of information following on the same line. Extra pieces of information on the command line are called *arguments*. They must be separated from the command name and from each other by one or more spaces. Sometimes an argument is a piece of text on which the command operates; alternatively, it can simply be some additional information about what the command is to do. For example,

  .sp 3

  shows an nroff command with one argument. The command requests vertical space; the argument indicates the number of blank lines desired.
- Finally, there is an important rule about the order in which things should appear in an input file to be processed using the –ms macros: *the file should not begin immediately with a line of text.* Instead, one of the following –ms macros must precede the first line of text:

  .TL .SH .NH .PP .LP

  Such a command placed before the beginning of text is called the *initializing macro* in the file.‡ The commands are described in the following pages. If none of them seems to be exactly what you want, use **.LP**.

2. Commands and Features of –ms

2.1. Paragraphs

An ordinary paragraph is produced by the command **.PP**, followed on subsequent lines by the text of the paragraph. In the output, it is set off by vertical space from whatever preceded it, and the first line is indented. This particular paragraph is an example of one produced by **.PP**. Another type of paragraph is produced by the command **.LP**, for "left-block paragraph." This also is preceded by vertical space when printed, but the first line is not indented. The amount of vertical spacing before paragraphs, and the width of the idention, normally have pre-defined standard or *default* values. Section 2.13 explains how to modify default features.

A third kind of paragraph is available by means of the command **.IP**, which stands for "indented paragraph." Here are some illustrations of the ways **.IP** might be used in an input file:

†The two control characters have slightly different meanings in a few cases; it is preferable to use the period except in rare instances when the alternate effect is needed. This point is explained in *A Troff Tutorial*. See "For More Information," section 7

‡This point is discussed more fully below, and sample beginnings of input files are shown in section 5.

```
.IP
The first example is simple.
This text will be set in a block
and the entire block will be indented
from the left margin.
.IP (2)
The second instance shows how to put
a hanging label on an indented
paragraph.
The text of the label is typed as an
argument to the .IP command.
The label will be aligned at the left
margin of the paper, while the
paragraph is indented
a standard amount.
.IP "Example 3" 12
A complication arises if the label
is too long to fit in the standard
indention provided by the command.
In this case,
you must request a non-standard
indention as a second argument
on the command line.
```

Now let's look at the way troff formats these paragraphs:

The first example is simple. This text will be set in a block and the entire block will be indented from the left margin.

(2) The second instance shows how to put a hanging label on an indented paragraph. The text of the label is typed as an argument to the .IP command. The label will be aligned at the left margin of the paper, while the paragraph is indented a standard amount.

Example 3 A complication arises if the label is too long to fit in the standard indention provided by the command. In this case, you must request a non-standard indention as a second argument on the command line.

The indention request in Example 3 is understood by -ms to be a number of *ens* of distance. An en is a unit of dimension used frequently in typesetting; we will discuss later what it means in nroff and troff. After an indented paragraph with non-standard indention, that indention stays in effect for a series of consecutive .IP's. It persists until the next .LP or .PP is used, at which point it it is reset to the standard amount. Also, the label "Example 3" must be enclosed within double-quote marks because it contains a space; otherwise, the space would signify the end of the argument.

Finally, the macro .QP produces a block-quote paragraph indented on both left and right, like this one. .QP's may be used in succession; the indention will not accumulate, but will remain constant for all of them.

2.2. Section Headings

Two varieties of section headings are available with -ms: unnumbered with .SH and automatically numbered with .NH. In either case, the text of the section heading is typed on one or more lines following the command. The end of the section heading is indicated by a subsequent paragraph command or by another section heading command. When printed, the heading is preceded by one line of vertical space and begins at the left margin. Nroff underlines the heading, while troff sets it in boldface type.

.NH section headings are numbered automatically. The macro takes an argument representing the *level-number* of the heading, up to 5. A third-level section number would be one like "1.2.1." The macro adds one to the section number at the requested level, as shown in the following example.

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

The input shown above generates the following output:

1. Erie-Lackawanna

1.1. Morris and Essex Division

1.1.1. Gladstone Branch

1.1.2. Montclair Branch

1.2. Boonton Line

2. Union Pacific

.NH without a level-number means the same thing as .NH 1. .NH 0 cancels the numbering sequence in effect and produces a section heading numbered 1.

2.3. Changes in Indention

The position of a paper's left margin is determined by two page-layout dimensions, the *page offset* and the *indention*. The page offset represents an absolute limit for the left margin, and usually is not changed at any point in the paper. The indention, on the other hand, controls the current left margin (the place where a section heading or a .LP paragraph begins), and this may be varied from one part of the paper to another. Indention is expressed as a distance to the right of the page offset. The indent is set by default to zero, i.e., the same point at which the page offset is placed. It cannot be moved to the left of the page offset.

Two macros, .RS and .RE, allow you to shift the indention of a paper to the right and back to the left, respectively. More than one .RS may be used to shift the indention a larger amount; to get back to the original margin, each must be balanced by an .RE. For example:

```
.IP I.
Branches of Government
.RS
.IP A.
Executive
.IP B.
Legislative
.RS
.IP 1.
House
.IP 2.
Senate
.RE
.IP C.
Judicial
.RE
```

produces:

I. Branches of Government
   A. Executive
   B. Legislative
      1. House
      2. Senate
   C. Judicial

2.4. Emphasis

Emphasis in typewritten material is usually indicated by underlining. In typesetting the convention is different; the usual way to emphasize words is to set them in a different typeface such as *italic* or **boldface**. In keeping with the design of compatibility between nroff and troff, -ms provides commands which emphasize text in a manner appropriate to the type of output.

The macro .I produces italics on the typesetter, underlining on the typewriter. .B gives boldface typesetting, underlined typewriting. A third macro, .R, restores the normal typeface or non-underlined typewriting. The commands are used this way:

```
.I
This text will be output
in italics (or underlined).
.R
Text here will be in normal
printing style.
.B
This is text to be set
in boldface (or underlined).
```

If only one word is needed in italics or boldface, it may be given as an argument on the command line, like the following:

```
.I word
or
.B word
```

In these examples, no .R is needed to restore normal printing for the following text. Also, when .I or .B is used with a word as an argument, it can take as a second argument any trailing punctuation to be printed immediately after the word but set in the normal typeface. For example,

```
.B word )
```

will print the word in boldface while the closing parenthesis will appear in the normal

typeface, directly adjacent to the word.

On the typesetter, actual underlining is available only in a limited way, by means of the .UL command. It is used like this:

```
.UL word
```

There is no way to underline more than one word at a time on the typesetter, except by repeating the .UL command for each word to be underlined.

2.5. Type Size Changes

Three macros control the size of type used for troff output. The command .LG increases the type size by two points, while .SM decreases it by two points. A *point* is another unit of dimension used commonly in typesetting; its precise meaning is discussed later (see section 2.13.1). Either command may be repeated for added effect, like this. The macro .NL restores the normal point size, cancelling all accumulated changes. These commands are useful primarily for temporary size changes for a small number of words. They do not affect vertical spacing of lines of text. Other techniques are available for changing the type size and vertical spacing of longer passages.† Commands for changing type size are ignored by nroff.

2.6. Boxes Around Text

A box can be drawn around a single word with the command .BX, which is used as follows:

```
This
.BX word
will appear in a box.
```

The output looks like this:

This word will appear in a box.

To get several lines of text enclosed in a box, precede the text with .B1 and follow it with .B2:

```
.B1
These boxes are designed to look good
when typeset, but aren't as pleasing
in typewriter output.
.B2
```

produces:

These boxes are designed to look good when typeset, but aren't as pleasing in typewriter output.

2.7. Title Pages and Cover Sheets

A group of macros is available to format items that typically appear on the cover sheet and/or title page of a paper. These commands generate a formally laid-out cover sheet and title page. It is possible to use them selectively (e.g., you might use the .TL command without the others); but if several are used, they should appear in the order shown below, normally at or near the beginning of the input file:

```
.RP (requests a cover sheet)
.TL
The title of the paper is typed here;
it may occupy one or more lines.
.AU
One or more author's names, arranged
on one line or multiple lines, as
you would like them to appear.
.AI
Information about the author's
institution, arranged on one
or more lines.
.AB
Abstract of the paper, a brief
description of its contents.
.AE (marks the end of the abstract)
```

If the macro .RP precedes .TL, the title, author and abstract material will be printed separately on a cover sheet. The title and author information (not the abstract) is then repeated automatically on page one (the title page) of the paper, without having to be typed again in the input file. In the absence of .RP, all of this material appears on page one, followed on the same page by the main text of the paper. The abstract is preceded by a centered heading of the word ABSTRACT. To suppress this heading, use the command .AB no instead of .AB. If there are several authors from different institutions, the names and institutions may be interleaved, with alternating .AU's and .AI's.

The use of the cover sheet and title page commands is entirely optional; you may begin a paper simply with a section heading

†See "Modifying Default Features," section 2.13.

or paragraph command. When cover sheet/title page material precedes the text, include a paragraph or section heading command between the last title page command and the beginning of the main text.†

2.8. Dates

When you use -ms, nroff normally prints the current date at the bottom center of every page starting with page one; troff does not. Both nroff and troff print it on the cover sheet if you have requested one with .RP. There are various ways of changing these defaults. To eliminate the date from nroff output, use the command .ND at the beginning of your input file. To make troff print the date as the center page footer, use .DA. To make nroff or troff print some date other than the actual current date, use .DA as follows:

.DA January 14, 1960

Finally, the command

.ND May 1, 1787

causes the specified date to be printed on the cover sheet, and nowhere else, when you use .RP. Place either .ND or .DA before the .RP.

Notice that in the two examples above, no double-quote marks were placed around the dates. These two commands represent exceptions to the rule that an argument containing spaces must be enclosed within double-quote marks.

2.9. Multi-column Formats

If you do not request otherwise, nroff produces output in single-column format. By placing the command .2C in your input file, you cause the output to be printed in double-column format beginning at that point. Each column will have a width 7/15 that of the text line length in single-column format, and the "gutter," or gap between columns, will be 1/15 of the full line length. To return to single-column, use the command .1C. Switching from double to single-column always causes a skip to a new page.

To obtain formats of more than two columns, use the command .MC as follows:

.MC *column-width*

This will cause output to be formatted in as many columns of the specified width as will fit on the page. The column-width may be specified in any unit of scale, but if no unit is indicated the setting will be understood as a number of ens.‡ .MC without any column-width specification means the same thing as .2C. Any change in the number of columns, except from one to a larger number, causes a skip to a new page.

2.10. Footnotes

The macros .FS and .FE indicate the beginning and end of material to be saved and printed at the bottom of the page as a footnote. Usage is as follows:

This sentence is in the
main body of text.*
.FS
***This is the footnote.**
.FE
Continuation of the text . . .

By default, footnotes are given a line length slightly shorter than the normal text, and, when typeset, appear in smaller size type. The commands only save a passage of text for printing at the bottom of the page; they do not mark the footnote reference in any way. Thus, in the example above, the asterisk had to be included as part of the text preceding the footnote, and again as part of the footnote text. Any character may be used as a footnote marker.

Warning: When including a footnote in your text, don't forget the .FE to mark the end. Failure to include this causes the rest of your text to be processed as if it were part of the footnote, resulting in one of several error conditions.

†Some sample beginnings of input files are shown in section 5, to help clarify the proper sequence of commands and text.

‡Units of scale are discussed in section 2.13.1.

*This is the footnote.

2.11. Keeping Text Together

The -ms package provides macros for keeping a block of text all together on one page. There are two ways of doing this. The standard "keep" is begun with the macro .KS and ended with .KE. If there is enough room on the current page for the material contained between these two macros, nroff prints it there; if not, it skips to the next page and prints it there instead. The other type, called a "floating keep," is begun with .KF and ended with .KE. If it is necessary to skip to a new page to print this material, nroff first fills the current page with the ordinary text that follows the keep in the input file. This avoids leaving blank space at the bottom of the page preceding the kept material. Typically, a floating keep would be useful for positioning a table or some other type of material not part of the strict logical sequence of text. It is essential to end every keep with .KE.

In double- or multi-column formats, the keep macros attempt to place all the kept material in the same column.

If the material enclosed within a keep turns out to require more than a page of space, or more than a column in multi-column format, it will start on a new page or column and simply run over onto the following page or column.

2.12. Displays

Occasionally it is desirable to format some text without filling and adjusting it—for example, a list of items or a stanza of poetry. To turn off filling so that each output line will correspond exactly to one line of input, use the command .DS to start the material and .DE to end it. By default, this material is indented from the left margin. Here is some sample input:

```
.DS
Display
text
lines
.DE
```

The resulting output is:

Display
text
lines

If you don't want the indention, use .DS L to begin and .DE to end, and

you'll get
something
like this.

A centered display begins with .DS C;

this is an
example
of a centered display.

Note that in the example above, each line is centered individually. To get a left-adjusted block that is

centered
on the page,
use .DS B to start.

Another possibilty is .DS I, which means the same thing as plain .DS. You can specify the amount of indention by including another argument after either of these constructions; .DS I 3 or .DS 3 begins a display to be indented 3 ens from the margin.

Any of the displays described above is automatically put into a standard keep. To avoid this, use the commands .CD, .BD, .LD or .ID instead of .DS C, .DS B, .DS L, or .DS I, respectively. Use .DE to end any type of display; failure to do this causes problems similar to those caused by failure to end a footnote or keep.

2.13. Modifying Default Features

One of the things -ms does to expedite document formatting is to establish a standard page layout style. In papers produced with -ms, the text line has a default length of six inches; the indention of the first line of a paragraph is five ens; the page number is printed at the top center of every page after page one; and so on. Many of these features are controlled by values stored by -ms as variables in the computer's memory. This makes it possible to alter the default format characteristics by changing the values that control them.

The memory locations where these values are stored are called *number registers* and *string registers*. Number and string registers have names like those of commands, one or two characters long. For

instance, the value of the line length is stored in a number register named LL. Unless you give a command to change the value stored in register LL, it will contain the standard or default value assigned to it by -ms. Table 2, below, lists the number registers you can change along with their default values.

2.13.1. Dimensions

In order to change a dimension like the line length from its default value, you can reset the associated number register. To do this, use the nroff command .nr as follows:

```
.nr LL 5i
```

The first argument is the name of a number register, and the second is the value being assigned to it. The value may be expressed as an integer or may contain a decimal fraction. When setting the value of a number register, it is almost always necessary to include a unit of scale immediately after the value. In the example above, the "i" as the unit of scale lets nroff know you mean five *inches* and not five of some other unit of distance. But the point size (PS) and vertical spacing (VS) registers are exceptions to this rule: ordinarily they should be assigned a value as a number of points *without indicating the unit of scale*. For example, to set the vertical spacing to 24 points, or one-third of an inch (double spacing), use the command

```
.nr VS 24
```

In the unusual case where you want to set the vertical spacing to more than half an inch (more than 36 points), include a unit of scale in setting the VS register. Table 1 explains the units of measurement available with nroff and troff.

The units *point*, *pica*, *em*, and *en* are units of measurement used by tradition in typesetting. The *vertical space* unit also corresponds to the typesetting term "leading," referring to the distance from the baseline of one line of type to the baseline of the next. Em and en are particularly interesting in that they are proportional to the type size currently in use (normally expressed as a number of points). An em is the distance equal to the number of points in the type size (roughly the width of the letter "m" in that point size), while an en is half that (about the width of the letter "n"). These units are convenient for specifying dimensions such as indention. In troff, em and en have their traditional meanings—i.e., one em of distance is equal to two ens. For nroff, on the other hand, em and en both mean the same quantity of distance, the width of one typewritten character.

Table 1
Units of Measurement in Nroff and Troff

Unit	*Abbr*	*Meaning For Nroff*	*Meaning For Troff*
point	p	1/72 inch	1/72 inch
pica	P	1/6 inch	1/6 inch
em	m	width of one character	distance equal to number of points in the current typesize
en	n	width of one character	half an em
vertical space	v	amount of space in which each line of text is set, measured baseline to baseline	
inch	i	inch	inch
centimeter	c	centimeter	centimeter
machine unit	u	1/240 inch	1/432 inch

The *machine unit* is a special unit of dimension used by nroff and troff internally. This is the unit to which the programs convert almost all dimensions when storing them in memory, and is included here primarily for completeness. In using the features of -ms described in this paper, it is sufficient to know that such a unit of measurement exists.

There is another important aspect of number registers. Because of the way -ms uses them, a change to a register such as LL does not immediately change the related dimension at that point in the output. Instead, in the case of the line length, the change takes place at the beginning of the next paragraph, where -ms resets various dimensions to the current values of the related number registers. Table 2 lists, for each register, the place at which a change to the register actually takes effect.

If the effect is needed immediately—if, for instance, you need to change the vertical spacing in the middle of a paragraph—you must use the nroff command .vs, which controls the vertical spacing directly. It

takes effect at the place where it occurs in your input file. Since it does not change the VS register, however, its effect lasts only until the beginning of the next paragraph. As a general rule: to make a permanent change, or one that will last for several paragraphs until you want to change it again, alter the value of the -ms register. If the change must happen immediately, somewhere other than the point shown in Table 2, use the nroff command.† If you want the change to be both immediate and lasting, do both.

Table 2
Summary of -ms Number Registers

Reg. Name	*Controls*	*Takes Effect*	*Default*
PS	point size	next para.	10
VS	vertical spacing	next para.	12
LL	line length of text	next para.	6i
LT	line length of running titles	next page	6i
FL	line length of footnotes	next FS	11/12 LL
PD	vertical offset of pargraphs	next para.	0.3v (troff) 1v (nroff)
PI	para. indent	next para.	5n
QI	left and right indent for QP	next QP	5n
PO	page offset	next page	26/27i(troff) 0 (nroff)
HM	top margin	next page	1i
FM	bottom margin	next page	1i

2.13.2. Page Headers and Footers

In setting up the default page layout, -ms provides for six *string registers* to store the running titles to be printed at tops and bottoms of pages. Like number registers, string registers are storage locations in the computer's memory; they differ in that their contents are strings of characters rather than numeric values. The -ms string register names are **LH**, **CH**, and **RH**, whose contents are printed in left-, center- and right-adjusted positions, respectively, at the top of every page after page one; and **LF**, **CF**, and **RF**, whose contents are printed at the bottom of every page starting with page one.

For nroff output, the default value of **CH** is the current page number surrounded on either side by hyphens; **CF** contains the current date as supplied by the computer. For troff, **CH** also contains the page number but **CF** is empty. The other four registers are empty by default for both nroff and troff. You can use the command **.ds** to assign a value to a string register. For example:

.ds RF Not for publication

This causes the character string "Not for publication" to be printed at the bottom right of every page. No double-quote marks are needed to enclose the argument; this is another exception to the rule about spaces within arguments. In order to remove the contents of a string register, simply redefine it as empty. For instance, to clear string register CH, and make the center header blank on following pages, use the command

.ds CH

To put the page number in the right header, use this command:

.ds RH %

In a string definition, "%" is a special symbol referring to nroff's automatic page counter. If you want hyphens on either side of the page number, place them on either side of the % in the command.

Commands that set the values of string and number registers, if they are meant to take effect as of the first page of output, should be placed at or near the beginning of the input file, before the initializing macro‡ (which, in turn, must precede the first line of text). Among other functions, the initializing macro causes what is called a "pseudo page break" onto page one of the paper, including the top-of-page processing for that page. It is particularly important that commands changing the value of the **PO**, **HM** and **FM** number registers and the page header string registers be placed before the transition onto the page where they are intended to take effect.

†See "For More Information," section 7.

‡The initializing macro can be the title macro or one of the paragraph or section heading macros. This point is discussed in section 1.3. Sample beginnings of input files are shown in section 5

2.14. Accent Marks

Certain foreign-language accent marks have been predefined as strings in the -ms package. Use them by placing a reference to the accent string before the letter being accented. The string reference *' placed immediately before the letter "e" in input text, as in

```
t\*'el\*'ephone
```

causes an acute accent to be placed over the "e" in the output:

téléphone

Here is a list of the accent strings with examples of their use:

Input	Output	Input	Output
*'e	é	*¯a	ā
*`e	è	*Ce	è
*:u	ü	*,c	ç
*^e	ê		

3. Using Nroff/Troff Commands

The -ms macros comprise a package in the sense that they are designed to meet most formatting needs, and to make it unnecessary to learn a large amount of detail about the more complex nroff command language. You can, however, use a small subset of nroff commands without losing too much of the macro package's simplicity. It is necessary to use the nroff commands **.nr** and **.ds** to manipulate the -ms number and string registers, as discussed previously. In addition, the following nroff commands may be used freely in a file to be processed using the -ms macro package:

- **.bp** begin a new page.
- **.br** break line: start a new output line whether or not the current one has been completely filled with text.
- **.sp** *n* provide *n* blank lines at this point in the output. If *n* is omitted, the command requests one blank line (the current value of the unit v). You can attach a unit of dimension to *n* to specify the quantity in units other than a number of blank lines.
- **.ce** *n* center the following *n* input text lines individually in the output. If *n* is omitted, only the next line of text is centered.
- **.na** turn off adjusting of right margin (ragged right).
- **.ad b** adjust both margins. This is the default adjust mode.

There is a reason for caution in using nroff commands in a file also containing -ms macros. The macro package executes sequences of nroff commands on its own, in a manner invisible to users. By inserting your own nroff commands you run the risk of introducing errors. The most likely unintended result is simply for your nroff commands to be ignored, but in some cases the results can include fatal nroff errors and expensive, garbled typesetter output.

For a very mild example, if you tried to produce a centered section heading with the input

```
.ce
.SH
Text of section heading
```

you would discover that the heading came out left-adjusted: the **.SH** macro, appearing after the **.ce** command, overrules it and forces left-adjusting. On the other hand, the sequence

```
.sp
.ce
.B
Line of text
```

would successfully produce a centered, boldface heading preceded by one line of vertical space. Because it is not possible to document in a simple way which tricks like this work and which don't, it is necessary to make the following generalization. Adaptations of more sophisticated features of the nroff language to files being processed with -ms, while possible, are not recommended for new or casual users of the document formatting programs.

4. Including Tables and Equations

UNIX provides special programs to make formatting of tables and mathematics relatively easy. These are *tbl* to format tables, and *eqn* and *neqn* to handle mathematics for typesetter and typewriter output, respectively. They each have their own command

languages, documented in separate write-ups.† Tables and equations are produced by "preprocessing" an nroff input file: the table and equation formatters convert material entered in their command languages to straight nroff input, and then nroff produces the tables or mathematics.

You can include tbl material in your nroff input file along with ordinary text (as was done in this paper to produce the tables on pages nine and ten). The way to identify this material is to precede each table with the command .TS and follow it with .TE. These have special meaning to the tbl preprocessor, signalling the beginning and end of material intended for it. If you are using the -ms package, the commands have additional significance as -ms macros. In a file processed with -ms, tables are set off by extra vertical space both before and after. Also, -ms offers a variant .TS H for beginning a table. This has a corresponding command, .TH, which is placed within the table data. All table text up to the .TH is used as a continuation heading if the table runs over onto more than one page.

For mathematics the situation is analogous. .EQ and .EN signal the beginning and end of material to be processed by eqn or neqn. When used with -ms, the .EQ and .EN commands cause the equation material to be formatted specially. .EQ can take one or two arguments. One is a format indicator: use .EQ I for an indented equation, .EQ L for left-adjusted, and .EQ C for centered (the same thing you get with just .EQ). The other possible argument is an equation number, which will be printed in the right margin alongside the equation. You can use either of these arguments without the other, but if both are used, the format indicator should come first: .EQ 3.1 specifies a centered equation numbered 3.1, while .EQ L 3.2 specifies a left-adjusted equation numbered 3.2.

5. Sample Input Files

Following are three sample files of input for nroff or troff. They are short, and are not meant to show the usage of all of the features covered in the previous sections. Rather, they focus on how to begin an input file. The order of occurrence of commands and text at the beginning of the file is important, and a number of problems can arise from errors at this point. After each example are some comments. No output from these samples is included here, but as an exercise you might try creating input files identical to the ones shown and then using nroff to produce output from them.

```
.ND
.nr LL 5.5i
.ds CH Hard Times - Chapter I
.ds CF - % -
.TL
Chapter I:
The One Thing Needful
.LP
Now, what I want is, Facts.
Teach these boys and girls nothing
but Facts.
Facts alone are wanted in life.
Plant nothing else, and root out
everything else.
You can only form the minds
of reasoning animals upon Facts:
nothing else will ever be of any
service to them.
```

The example above is a fairly simple one. When the file contains commands that set values of number and string registers, and the effect is wanted at the beginning of the output, those commands should come first. Their order relative to each other is not important. The commands .ND or .DA, if used, should also be in this group appearing first. Next comes the initializing macro, which in this case is the .TL command. The paragraph command, .LP, signifies the end of the title and the beginning of the main text of the paper.

The second example is a bit more complex:

```
.ND
.nr LL 5.5i
.nr QI 10n
.ds LH Gulliver's Travels
.ds CH
.ds RH Lilliput
.ds CF - % -
.LP
.ce
.B
```

†See "For More Information," section 7

```
.LG
Chapter VIII
.NL
.QP
.I
The author, by a lucky accident,
finds means to leave Blefuscu;
and, after some difficulties,
returns safe to his native country.
.R
.LP
Three days after my arrival, walking
out of curiosity to the north-east coast
of the Island I observed, about half
a league off, in the sea, somewhat that
looked like a boat overturned.
```

This example shows a typical way of specifying a title by means other than the .TL command. The reason for using an alternate method is to avoid the standard vertical placement of the title provided by .TL, and simply center it at the top of the page. The subtlety here is that, because .TL is not being used, there must be another macro to perform initialization. In this case it is .LP, included solely for this purpose, even though the following line of text is a title and not the beginning of a paragraph.

Finally, the third example shows the usage of the cover sheet macros at the beginning of a file.

```
.ND
.ds CH
.ds CF - % -
.nr PS 9
.nr VS 11
.nr LL 5i
.nr PI 3n
.RP
.TL
Confessions of an
English Opium-Eater
.AU
Thomas De Quincey
.AB no
I here present you, courteous reader,
with the record of a
remarkable period of my life;
and according to my application of it,
I trust that it will prove,
not merely an interesting record, but
in a considerable degree, instructive.
.AE
.MC 2.3i
.PP
I have often been asked how it was,
and through what series of steps,
that I became an opium-eater.
Was it gradually,
tentatively, mistrustingly,
as one goes down a shelving beach into
a deepening sea, and
with knowledge from the first of the
dangers lying on that path;
half-courting those dangers, in fact,
whilst seeming to defy them?
```

When cover sheet / title page macros are used, there are more things to keep in proper order. The commands that change number and string registers still come first. Next should be the .RP command, if a cover sheet is desired. Following this are the commands and text for the elements of the cover sheet and title page. It is not necessary to include all of them, but any that you use should be in the order shown in section 2.7. If an abstract is included, don't forget the .AE to end the abstract. After the cover sheet or title page material there must be a section heading or paragraph macro, after which begins the main body of text. If multi-column format is wanted for the main text, a .2C or .MC command may be placed between title page material and the paragraph or section-heading command.

6. Producing Output

When you have finished preparing the input file for a document, you are ready to produce the formatted output. This is done by means of the UNIX commands **nroff** or **troff**. Typically, you might wish to preview the output for typographical errors and mistakes in formatting. There are ways to preview either nroff or troff on crt (video screen) terminals, typewriter terminals, or the line printer, whichever is most convenient and appropriate. In addition, troff output can be previewed on a Tektronix 4015 graphics terminal, providing a reasonable facsimile of phototypesetter output.

The general form of the command to produce output is

nroff (or **troff**) *options filename ...*

The *options* are described fully in the writeup on nroff and troff in section 1 of the *UNIX Programmer's Manual*. We can only touch on them here, although one should get special mention. In the command

nroff -ms *filename* ...

the -ms option has the effect of informing nroff that you are using the -ms macro package. If you forget this option, you get continuous, unpaginated output in which -ms macro commands are ignored.

More than one input file can be named in the command line (as indicated by the ellipses after *filename*), in which case nroff simply processes all of them, in the order they appear, as if they were all one file.

Following are some examples of commands you might use to get preview and final output of various sorts. Send nroff output to lineprinter:

nroff -ms -Tlpr *filename* ... | **lpr**

Stop after each page to change paper on a typewriter terminal (type "control-d" as signal to resume output when ready):

nroff -ms -T*type* **-s1** *filename* ...

(*Type* in the command line above refers to a code for the type of terminal being used—see "For More Information," section 7, for a reference on this.)

Produce a file with tables (preprocess with *tbl*):

tbl *filename* ... | **nroff -ms -T***type*

Produce a file with equations (preprocess with *neqn*):

neqn *filename* ... | **nroff -ms -T***type*

Produce a file with tables and equations (should be done in this order):

tbl *filename* ... | **neqn** | **nroff -ms -T***type*

Put a job in the phototypesetter queue:

troff -Q -ms *filename* ...

Typeset a file with tables and equations (note that eqn is used with troff, whereas neqn is used with nroff):

tbl *filename* ... | **eqn** | **troff -ms -Q**

Preview a troff approximation on terminal (may use tbl and/or eqn as above):

troff -a -ms *filename* ...

Print a troff preview on lineprinter:

troff -a -ms *filename* ... | **lpr**

Preview a troff approximation on a Tektronix 4015 graphics terminal (may use tbl and/or eqn as above):

troff -t -ms *filename* ... | **tc**

7. For More Information

This document is intended to be the main written source of information on formatting ordinary text with nroff or troff and the -ms macro package. Other documents contain information not covered here. If your text contains tables or mathematics, you should consult the separate manuals describing the programs that work in conjunction with nroff and troff to format that material: *Tbl—A Program to Format Tables* by M. E. Lesk, and *Typesetting Mathematics—User's Guide* by Brian W. Kernighan and Lorinda L. Cherry. *A Troff Tutorial* by Brian W. Kernighan provides very useful supplementary information on the nroff and troff command language, though it covers many features of these programs that should be used only with caution in a file to be processed with -ms. The comprehensive reference source, the *Nroff/Troff User's Manual* by Joseph F. Ossanna, is difficult to read and recommended mainly for prospective experts.

When you are ready to produce formatted output, consult the *UNIX Programmer's Manual* pages on nroff and troff for details on command usage and the various command-line options. The manual writeup on eqn(1) should also be consulted by users of eqn. Writeups from the Programmer's Manual are available online by means of the **man** command; the whole Manual is available in printed form as well. A one-page document entitled *Using Hardcopy Terminals with Nroff* contains tips for producing nroff output on typewriter terminals such as the DTC 302, IPSI 1622, and others. This includes instructions on how to set up the terminal, and a list of identifiers for various terminal types.

Other writeups are available online by means of the **help** command. Type **help**

index for a list of topics.

All of the printed documentation mentioned here is available at the Computing Services Library, 218 Evans Hall. For further information, please drop by the Consulting Office or phone the UNIX consultant at 642-4072.

Appendix A: Command Descriptions

This appendix serves as a reference manual for the -ms macro package; the intention here is to provide a concise but complete description of the operation of each command. Certain typographical conventions are used in presenting command syntax. Command names appear at the left margin, followed where appropriate by the arguments available with the command. Arguments are typed on the same line as the command, separated from the command and from each other by a space. An argument which contains one or more spaces within it must be surrounded by double-quote marks except where noted. **Boldface** indicates what must be typed literally as shown in the syntax statement; thus each command name appears in boldface. A word in *italics* represents an argument which you supply. The contents of the argument are sometimes entirely your choice (for example, the *label* after the command **.IP** can be anything). Sometimes the argument is restricted to a predetermined set of choices (for example, the *level-number* after the **.NH** command must be an integer from 0 to 5). Details about what can be supplied as an argument are contained in the description opposite each command.

An argument enclosed in square brackets is optional—the command has meaning either with or without that argument. Conversely, an argument not enclosed in square brackets must be supplied whenever the command is used. An argument enclosed in square brackets and printed in boldface is optional, but, if used, must be typed literally as shown.

Paragraphs

.PP

Begins a standard paragraph, separated vertically from preceding text by the value of number register PD. First line is indented by the value of register PI, and following lines begin at the current main indent level.

.LP

Begins a left-block paragraph, set off vertically by the value of register PD. No first-line indention. All lines begin at the main indent level.

.IP [*label*] [*indent*]

Begins an indented paragraph, set off vertically by the value of register PD. The entire block is left-adjusted and then, by default, indented the value of register PI to the right of the main indent level. If one argument is given, it is a label to be placed at the main indent level opposite the first line of the paragraph. If two arguments are given, the second must be numeric and is an amount of indention (in ens unless indicated otherwise) to supercede the default indention for the paragraph. Nonstandard indention must be specified if the label is too wide to fit within the default indention. This nonstandard indent persists in subsequent .IP's in a series, disappearing when the series is ended by a return to some other format such as a section heading or a .PP or .LP paragraph.

For nonstandard indention without any label, the first argument should be simply a pair of double-quote marks with nothing between them.

.QP

Begins a block-quote paragraph. Preceded by vertical space as for other paragraphs. Every line is indented from the main indent level by the value of register QI. The right margin is moved in toward the left by an equal amount (the line length is shortened). Successive .QP's maintain the same indention; it does not accumulate.

Section Headings

.SH — Begins a heading that is left-adjusted at the main indention level and separated by one vertical space from whatever preceded it. In nroff, the heading is underlined; in troff, it is set in boldface.

.NH [*level-number*] — Produces a heading similar to .SH except that it is automatically given a consecutive number. The optional level number, from 1 to 5, causes the macro to generate the next consecutive section number of that level (eg., 1.2.5 is a third-level section number). A level-number 0 (zero) may be used as the argument; this cancels the numbering sequence in effect and generates a heading numbered 1.

Note: When either .SH or .NH is used, all text up to the next paragraph command or section heading command is considered part of the heading.

Changes in Indention

.RS — Moves the indention to the right by a value based on register PI. More than one .RS may be used, producing additional indention.

.RE — Moves the indention to the left by the same amount as the corresponding .RS moved it to the right. To restore the original indent, each .RS must be balanced by a corresponding .RE.

Notes: it is not possible to move the indent level to the left of the page offset. The value of register PI should not be changed within a series of .RS and .RE commands at any point except after the indention has been returned to its default starting position.

Emphasis and Size Changes

.I [*word*] [*punctuation*] — Without an argument, this macro causes a switch to font number 2 (italic) in troff or underlined typing in nroff. If one argument is given, it is one word to be italicized, and the effect of the command is limited to that word. A second argument may consist of trailing punctuation to be printed directly after the word, in the typeface (usually roman) in use for the text prior to the italicized word.

.B [*word*] [*punctuation*] — Produces text in font number 3 (boldface) in troff, underlining in nroff. Usage is analogous to that of .I.

.R — Switches back to font number 1 (roman) in troff, non-underlined typing in nroff.

.UL *word* — Causes the word supplied as the argument to be underlined. This is the only –ms command to produce an underlined word on the typesetter. It works only for one word at a time.

.LG — Increases the type size by two points in troff. (May be repeated for added effect.) Ignored by nroff.

.SM — Reduces the point size by two points. May be repeated for added effect. Ignored by nroff.

.NL — Resets the point size to the normal setting, i.e., the value of the PS register. Ignored by nroff.

Note: if changing the type size by two points results in a non-existent type size on the typesetter, the next larger valid size is chosen. Valid point sizes are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36.

Boxes Around Text

.BX *word*	Draws a box around *word*.
.B1	Begins a longer passage of text to be enclosed in a box.
.B2	Ends passage of text and draws box.

Title Pages and Cover Sheets

.RP	Causes a cover sheet to be generated containing any of the following information, if included with the appropriate macro after .RP: title, authors, authors' institutions, and abstract. The current date is printed on the cover sheet unless suppressed with the command .ND.
.TL	When used for cover sheet and/or title page (prior to regular text), .TL causes title text to be filled, without hyphenation, on a 5-inch line length. The resulting lines are individually centered when printed. To break lines of title text differently, use the command .br. Troff sets the title in 12-point boldface.
.AU	Centers the author's name, included on the following line of the input file. More than one name can be included, in which case they will be printed on separate lines if entered on separate input lines. Troff sets names in 10-point italic.
.AI	Centers lines of information about the author's institution. .AU and .AI commands can be repeated, if dersired, for multiple authors from different institutions.
.AB [no]	Begins the abstract. When printed, this is preceded by a centered heading of the word ABSTRACT unless suppressed by use of the argument **no**. The abstract is filled, hyphenated and adjusted on a line length 5/6 the normal text line length.
.AE	Ends the abstract.
	Additional notes: In order for cover sheet/title page material to be handled properly, it must be followed by a macro such as one of the paragraph or section heading commands before the regular text begins. If .RP is used, all of the title/author/abstract material is put on the cover sheet and all except the abstract is repeated at the top of page one. Otherwise, all of the material is placed on page one prior to the beginning of regular text.

Dates

.ND [*date*]	When used without an argument, this macro suppresses printing of the date on the document. (By default, if .ND is omitted, -ms causes nroff to print the current date at the bottom center of every page, and on the cover sheet in .RP format; with troff, the date is printed only on the cover sheet.) If a date is given as an argument to this macro, it appears on the cover sheet in .RP format but nowhere else.
.DA [*date*]	Without an argument, DA causes the current date to be printed at the bottom of every page of output in troff, as well as on the cover sheet. (This is the default condition in nroff.) With a date as an argument, the command causes the specified date (rather than the current date) to be printed at the bottom of every page, and on the cover sheet, for both nroff and troff.

Note: when typing the date as an argument to either of these macros, you can include spaces without having to enclose the whole thing in double-quote marks as you ordinarily would in an argument to a command.

Multi-column Formats

.2C Switches to two-column format. Column widths are 7/15 of the current value of the LL number register; gutter width is 1/15 LL.

.1C Switches to single-column format (the default format). A switch from two or more columns to single-column causes a page break before output is resumed.

.MC [*column-width*] Switches to multi-column format. The number of columns is computed automatically: it will be the largest number of the specified width that can fit within the regular line length (the value of register LL). The column-width argument must be numeric (it may be an integer or contain a decimal fraction), and is understood to be a number of ens unless a different unit is indicated. If no column-width is specified, .MC means the same as .2C. Any change in the number of columns, except from one to a larger number, causes a page break first.

Footnotes

.FS Begins text of footnote. This macro, and the accompanying footnote text, should be placed in the input file immediately after the reference to the footnote. Footnote text is automatically saved and printed at the bottom of the current page, separated from the main text by a horizontal rule. If not enough space remains on the page for all of the footnote, it continues at the bottom of the following page. The line length of footnotes defaults to 11/12 of the normal line length (in multi-column output, this means 11/12 of the column width). In troff output, footnotes are set in 8-point type.

.FE Marks the end of footnote text.

Keeps

.KS Begins a standard keep. Text on following input lines will be kept together on one page if possible. If not enough space remains on the current page, a new page is begun at this point.

.KF Begins a floating keep. If not enough space remains on the current page for the keep, the current page will be completed with the input text that follows the end of the keep; the kept material then begins the next page.

.KE Marks the end of either standard or floating keep.

Note: In formats of two or more columns, the effect is to try to keep the material together in one column; if there isn't room in the current column, the material starts in the next.

Displays

.DS [*format*] [*indent*] Begins a display, i.e., unfilled text. Set off by vertical space before and after the display (1v before and after in nroff; 0.5v in troff). A format indicator may be given as an argument. The possible format indicators are:

- **L** left-adjusted
- **I** indented 0.5i (troff) or 8n (nroff)
- **C** each line is centered individually
- **B** left-adjusted lines are centered as a group

.DS with no format indicator means the same as .DS I. Either of these forms may also take a numeric argument representing a non-standard indention in ens. Any of the displays described above automatically invokes a keep.

.LD Left-adjusted display without invoking keep.

.ID [*indent*] Indented display without keep. Default indention is the same as for .DS I. Other indention may be specified as an argument.

.CD Lines individually centered, without keep.

.BD Left adjusted and then centered, without keep.

.DE Marks the end of any type of display.

Tables and Equations

.TS [H] Signals the beginning of material to be preprocessed by *tbl*. When used with -ms, it also has the effect of supplying half of a vertical space separation between the table and any preceding text. When used with the argument "H," table data up to the command ".TH" is understood as a running head for the table and recurs on following pages of a multi-page table. (This effect is obtainable only when -ms is used.)

.TH Signals the end of the running table heading.

.TE Signals the end of material to be preprocessed by tbl, and, with -ms, supplies half of a vertical space at the end of the table.

.EQ [*format*] [*number*] Marks the beginning of material to be preprocessed by *eqn* or *neqn*. When used with -ms, generates vertical separation before the equation is output and, by default, centers the equation in the output line. Placement of the equation can be controlled by use of a format indicator as an argument: use .EQ L for left-adjusted, .EQ I for indented, and .EQ C for centered. An equation number, whatever you choose, may also be given as an argument to .EQ. If both arguments are used, the format indicator should be placed first.

.EN Signals the end of material to be preprocessed by eqn or neqn.

Appendix B: Names of -ms Macros, String Registers, and Number Registers

The following macro, string register, and number register names are used by -ms internally. This list is useful primarily to those who define their own macros. Note that no lower case letters are used in any -ms internal name.

Macro and String Register Names

'	AB	C1	DE	EQ	I	KQ	ND	R	RT	TM
`	AE	C2	DS	EZ	I1	KS	NH	R1	S0	TQ
^	AI	CA	DW	FA	I2	LB	NL	R2	S1	TS
-	AT	CB	DY	FE	I3	LD	NP	R3	S2	TT
:	AU	CC	E1	FJ	I4	LG	OD	R4	SG	UL
_	B	CD	E2	FK	I5	LP	OK	R5	SH	US
1C	B1	CF	E3	FN	ID	MC	PP	RC	SM	UX
2C	B2	CH	E4	FO	IE	ME	PT	RE	SN	WB
A1	BB	CM	E5	FQ	IM	MF	PY	RF	SY	WH
A2	BG	CS	EE	FS	IP	MH	QE	RH	TA	WT
A3	BT	CT	EL	FV	IZ	MN	QF	RP	TE	XD
A4	BX	D	EM	FY	KE	MO	QP	RQ	TH	XF
A5	C	DA	EN	HO	KF	MR	QS	RS	TL	XK

Number Register Names

:	BI	FL	H4	IP	LL	NC	OI	PO	ST	TV
#T	BW	FP	HM	IR	MF	NF	PD	PS	TB	WF
1T	CW	GW	HT	IT	MM	NQ	PE	PX	TC	YE
AV	DW	H1	IF	KI	MN	NR	PF	QI	TD	YY
BE	EF	H2	IK	L1	MO	NS	PI	QP	TN	ZN
BH	FC	H3	IM	LE	NA	NX	PN	RO	TQ	

A TROFF Tutorial

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

troff is a text-formatting program for driving the Graphic Systems phototypesetter on the UNIX† and GCOS operating systems. This device is capable of producing high quality text; this paper is an example of **troff** output.

The phototypesetter itself normally runs with four fonts, containing roman, italic and bold letters (as on this page), a full greek alphabet, and a substantial number of special characters and mathematical symbols. Characters can be printed in a range of sizes, and placed anywhere on the page.

troff allows the user full control over fonts, sizes, and character positions, as well as the usual features of a formatter — right-margin justification, automatic hyphenation, page titling and numbering, and so on. It also provides macros, arithmetic variables and operations, and conditional testing, for complicated formatting tasks.

This document is an introduction to the most basic use of **troff**. It presents just enough information to enable the user to do simple formatting tasks like making viewgraphs, and to make incremental changes to existing packages of **troff** commands. In most respects, the UNIX formatter **nroff** is identical to **troff**, so this document also serves as a tutorial on **nroff**.

August 4, 1978

†UNIX is a Trademark of Bell Laboratories.

A TROFF Tutorial

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

troff [1] is a text-formatting program, written by J. F. Ossanna, for producing high-quality printed output from the phototypesetter on the UNIX and GCOS operating systems. This document is an example of **troff** output.

The single most important rule of using **troff** is not to use it directly, but through some intermediary. In many ways, **troff** resembles an assembly language — a remarkably powerful and flexible one — but nonetheless such that many operations must be specified at a level of detail and in a form that is too hard for most people to use effectively.

For two special applications, there are programs that provide an interface to **troff** for the majority of users. **eqn** [2] provides an easy to learn language for typesetting mathematics; the **eqn** user need know no **troff** whatsoever to typeset mathematics. **tbl** [3] provides the same convenience for producing tables of arbitrary complexity.

For producing straight text (which may well contain mathematics or tables), there are a number of 'macro packages' that define formatting rules and operations for specific styles of documents, and reduce the amount of direct contact with **troff**. In particular, the '−ms' [4] and PWB/MM [5] packages for Bell Labs internal memoranda and external papers provide most of the facilities needed for a wide range of document preparation. (This memo was prepared with '−ms'.) There are also packages for viewgraphs, for simulating the older **roff** formatters on UNIX and GCOS, and for other special applications. Typically you will find these packages easier to use than **troff** once you get beyond the most trivial operations; you should always consider them first.

In the few cases where existing packages don't do the whole job, the solution is *not* to write an entirely new set of **troff** instructions from scratch, but to make small changes to adapt packages that already exist.

In accordance with this philosophy of letting someone else do the work, the part of **troff** described here is only a small part of the whole, although it tries to concentrate on the more useful parts. In any case, there is no attempt to be complete. Rather, the emphasis is on showing how to do simple things, and how to make incremental changes to what already exists. The contents of the remaining sections are:

The **troff** described here is the C-language version running on UNIX at Murray Hill, as documented in [1].

To use **troff** you have to prepare not only the actual text you want printed, but some information that tells *how* you want it printed. (Readers who use **roff** will find the approach familiar.) For **troff** the text and the formatting information are often intertwined quite intimately. Most commands to **troff** are placed on a line separate from the text itself, beginning with a period (one command per line). For example,

```
Some text.
.ps 14
Some more text.
```

will change the 'point size', that is, the size of the letters being printed, to '14 point' (one point is 1/72 inch) like this:

Some text. Some more text.

Occasionally, though, something special occurs in the middle of a line — to produce

Area = πr^2

you have to type

```
Area = \(*p\fIr\fR\|\s8\u2\d\s0
```

(which we will explain shortly). The backslash character \ is used to introduce **troff** commands and special characters within a line of text.

2. Point Sizes; Line Spacing

As mentioned above, the command .ps sets the point size. One point is 1/72 inch, so 6-point characters are at most 1/12 inch high, and 36-point characters are ½ inch. There are 15 point sizes, listed below.

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

If the number after .ps is not one of these legal sizes, it is rounded up to the next valid value, with a maximum of 36. If no number follows .ps, **troff** reverts to the previous size, whatever it was. **troff** begins with point size 10, which is usually fine. This document is in 9 point.

The point size can also be changed in the middle of a line or even a word with the in-line command \s. To produce

UNIX runs on a PDP-11/45

type

```
\s8UNIX\s10 runs on a \s8PDP-\s1011/45
```

As above, \s should be followed by a legal point size, except that \s0 causes the size to revert to its previous value. Notice that \s1011 can be understood correctly as 'size 10, followed by an 11', if the size is legal, but not otherwise. Be cautious with similar constructions.

Relative size changes are also legal and useful:

```
\s-2UNIX\s+2
```

temporarily decreases the size, whatever it is, by two points, then restores it. Relative size changes have the advantage that the size difference is independent of the starting size of the document. The amount of the relative change is restricted to a single digit.

The other parameter that determines what the type looks like is the spacing between lines, which is set independently of the point size. Vertical spacing is measured from the bottom of one line to the bottom of the next. The command to control vertical spacing is **.vs**. For running text, it is usually best to set the vertical spacing about 20% bigger than the character size. For example, so far in this document, we have used "9 on 11", that is,

```
.ps 9
.vs 11p
```

If we changed to

```
.ps 9
.vs 9p
```

the running text would look like this. After a few lines, you will agree it looks a little cramped. The right vertical spacing is partly a matter of taste, depending on how much text you want to squeeze into a given space, and partly a matter of traditional printing style. By default, **troff** uses 10 on 12.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. This is 12 on 14.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. For example, 10 on 12 uses about twice as much space as 7 on 8. This is 6 on 7, which is even smaller. It packs a lot more words per line, but you can go blind trying to read it.

When used without arguments, **.ps** and **.vs** revert to the previous size and vertical spacing respectively.

The command **.sp** is used to get extra vertical space. Unadorned, it gives you one extra blank line (one **.vs**, whatever that has been set to). Typically, that's more or less than you want, so **.sp** can be followed by information about how much space you want —

```
.sp 2i
```

means 'two inches of vertical space'.

```
.sp 2p
```

means 'two points of vertical space'; and

```
.sp 2
```

means 'two vertical spaces' — two of whatever

.vs is set to (this can also be made explicit with .sp 2v); troff also understands decimal fractions in most places, so

```
.sp 1.5i
```

is a space of 1.5 inches. These same scale factors can be used after .vs to define line spacing, and in fact after most commands that deal with physical dimensions.

It should be noted that all size numbers are converted internally to 'machine units', which are 1/432 inch (1/6 point). For most purposes, this is enough resolution that you don't have to worry about the accuracy of the representation. The situation is not quite so good vertically, where resolution is 1/144 inch (1/2 point).

3. Fonts and Special Characters

troff and the typesetter allow four different fonts at any one time. Normally three fonts (Times roman, italic and bold) and one collection of special characters are permanently mounted.

abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

The greek, mathematical symbols and miscellany of the special font are listed in Appendix A.

troff prints in roman unless told otherwise. To switch into bold, use the .ft command

```
.ft B
```

and for italics,

```
.ft I
```

To return to roman, use .ft R; to return to the previous font, whatever it was, use either .ft P or just .ft. The 'underline' command

```
.ul
```

causes the next input line to print in italics. .ul can be followed by a count to indicate that more than one line is to be italicized.

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

bold*face* text

is produced by

```
\fBbold\fIface\fR text
```

If you want to do this so the previous font, whatever it was, is left undisturbed, insert extra \fP commands, like this:

```
\fBbold\fP\fIface\fP\fR text\fP
```

Because only the immediately previous font is remembered, you have to restore the previous font after each change or you can lose it. The same is true of .ps and .vs when used without an argument.

There are other fonts available besides the standard set, although you can still use only four at any given time. The command .fp tells troff what fonts are physically mounted on the typesetter:

```
.fp 3 H
```

says that the Helvetica font is mounted on position 3. (For a complete list of fonts and what they look like, see the troff manual.) Appropriate .fp commands should appear at the beginning of your document if you do not use the standard fonts.

It is possible to make a document relatively independent of the actual fonts used to print it by using font numbers instead of names; for example, \f3 and .ft 3 mean 'whatever font is mounted at position 3', and thus work for any setting. Normal settings are roman font on 1, italic on 2, bold on 3, and special on 4.

There is also a way to get 'synthetic' bold fonts by overstriking letters with a slight offset. Look at the .bd command in [1].

Special characters have four-character names beginning with \(, and they may be inserted anywhere. For example,

```
1/4 + 1/2 = 3/4
```

is produced by

```
\(14 + \(12 = \(34
```

In particular, greek letters are all of the form \(*—, where — is an upper or lower case roman letter reminiscent of the greek. Thus to get

$$\Sigma(\alpha \times \beta) \rightarrow \infty$$

in bare troff we have to type

```
\(*S(\(*a\(mu\(*b) \(-> \(if
```

That line is unscrambled as follows:

\(*S	Σ
(	(
\(*a	α
\(mu	×
\(*b	β
)	)
\(->	→
\(if	∞

A complete list of these special names occurs in Appendix A.

In **eqn** [2] the same effect can be achieved with the input

```
SIGMA ( alpha times beta ) -> inf
```

which is less concise, but clearer to the uninitiated.

Notice that each four-character name is a single character as far as **troff** is concerned — the 'translate' command

```
.tr \(mi\(em
```

is perfectly clear, meaning

```
.tr --
```

that is, to translate — into —.

Some characters are automatically translated into others: grave ` and acute ' accents (apostrophes) become open and close single quotes '"'; the combination of "..." is generally preferable to the double quotes "...". Similarly a typed minus sign becomes a hyphen -. To print an explicit — sign, use \-. To get a backslash printed, use \e.

4. Indents and Line Lengths

troff starts with a line length of 6.5 inches, too wide for 8½×11 paper. To reset the line length, use the .ll command, as in

```
.ll 6i
```

As with .sp, the actual length can be specified in several ways; inches are probably the most intuitive.

The maximum line length provided by the typesetter is 7.5 inches, by the way. To use the full width, you will have to reset the default physical left margin ("page offset"), which is normally slightly less than one inch from the left edge of the paper. This is done by the .po command.

```
.po 0
```

sets the offset as far to the left as it will go.

The indent command .in causes the left margin to be indented by some specified amount from the page offset. If we use .in to move the left margin in, and .ll to move the right margin to the left, we can make offset blocks of text:

```
.in 0.3i
.ll -0.3i
text to be set into a block
.ll +0.3i
.in -0.3i
```

will create a block that looks like this:

> Pater noster qui est in caelis sanctificetur nomen tuum; adveniat regnum tuum; fiat voluntas tua, sicut in caelo, et in terra. ... Amen.

Notice the use of '+' and '−' to specify the amount of change. These change the previous setting by the specified amount, rather than just overriding it. The distinction is quite important: .ll +1i makes lines one inch longer; .ll 1i makes them one inch *long*.

With .in, .ll and .po, the previous value is used if no argument is specified.

To indent a single line, use the 'temporary indent' command .ti. For example, all paragraphs in this memo effectively begin with the command

```
.ti 3
```

Three of what? The default unit for .ti, as for most horizontally oriented commands (.ll, .in, .po), is ems; an em is roughly the width of the letter 'm' in the current point size. (Precisely, a em in size *p* is *p* points.) Although inches are usually clearer than ems to people who don't set type for a living, ems have a place: they are a measure of size that is proportional to the current point size. If you want to make text that keeps its proportions regardless of point size, you should use ems for all dimensions. Ems can be specified as scale factors directly, as in .ti 2.5m.

Lines can also be indented negatively if the indent is already positive:

```
.ti -0.3i
```

causes the next line to be moved back three tenths of an inch. Thus to make a decorative initial capital, we indent the whole paragraph, then move the letter 'P' back with a .ti command:

> Pater noster qui est in caelis sanctificetur nomen tuum; adveniat regnum tuum; fiat voluntas tua, sicut in caelo, et in terra. ... Amen.

Of course, there is also some trickery to make the 'P' bigger (just a '\s36P\s0'), and to move it down from its normal position (see the section on local motions).

5. Tabs

Tabs (the ASCII 'horizontal tab' character) can be used to produce output in columns, or to set the horizontal position of output. Typically tabs are used only in unfilled text. Tab stops are set by default every half inch from the current indent, but can be changed by the .ta command. To set stops every inch, for example,

```
.ta 1i 2i 3i 4i 5i 6i
```

Unfortunately the stops are left-justified only (as on a typewriter), so lining up columns of right-justified numbers can be painful. If you have many numbers, or if you need more complicated table layout, *don't* use **troff** directly; use the **tbl** program described in [3].

For a handful of numeric columns, you can do it this way: Precede every number by enough blanks to make it line up when typed.

```
.nf
.ta 1i 2i 3i
  1 tab   2 tab   3
 40 tab  50 tab  60
700 tab 800 tab 900
.fi
```

Then change each leading blank into the string \0. This is a character that does not print, but that has the same width as a digit. When printed, this will produce

1	2	3
40	50	60
700	800	900

It is also possible to fill up tabbed-over space with some character other than blanks by setting the 'tab replacement character' with the .tc command:

```
.ta 1.5i 2.5i
.tc \(ru        (\(ru is "_")
Name tab Age tab
```

produces

Name _____ Age _____

To reset the tab replacement character to a blank, use .tc with no argument. (Lines can also be drawn with the \l command, described in Section 6.)

troff also provides a very general mechanism called 'fields' for setting up complicated columns. (This is used by **tbl**). We will not go into it in this paper.

6. Local Motions: Drawing lines and characters

Remember 'Area = πr^2' and the big 'P' in the Paternoster. How are they done? **troff** provides a host of commands for placing characters of any size at any place. You can use them to draw special characters or to tune your output for a particular appearance. Most of these commands are straightforward, but messy to read and tough to type correctly.

If you won't use **eqn**, subscripts and superscripts are most easily done with the half-line local motions \u and \d. To go back up the page half a point-size, insert a \u at the desired place; to go down, insert a \d. (\u and \d should always be used in pairs, as explained below.) Thus

```
Area = \(*pr\u2\d
```

produces

Area = πr^2

To make the '2' smaller, bracket it with \s-2...\s0. Since \u and \d refer to the current point size, be sure to put them either both inside or both outside the size changes, or you will get an unbalanced vertical motion.

Sometimes the space given by \u and \d isn't the right amount. The \v command can be used to request an arbitrary amount of vertical motion. The in-line command

```
\v'(amount)'
```

causes motion up or down the page by the amount specified in '(amount)'. For example, to move the 'P' down, we used

```
.in +0.6i          (move paragraph in)
.ll -0.3i          (shorten lines)
.ti -0.3i          (move P back)
\v'2'\s36P\s0\v'-2'ater noster qui est
in caelis ...
```

A minus sign causes upward motion, while no sign or a plus sign means down the page. Thus \v'-2' causes an upward vertical motion of two line spaces.

There are many other ways to specify the amount of motion —

```
\v'0.1i'
\v'3p'
\v'-0.5m'
```

and so on are all legal. Notice that the scale specifier i or p or m goes inside the quotes. Any character can be used in place of the quotes; this is also true of all other **troff** commands described in this section.

Since **troff** does not take within-the-line vertical motions into account when figuring out where it is on the page, output lines can have unexpected positions if the left and right ends aren't at the same vertical position. Thus \v, like \u and \d, should always balance upward vertical motion in a line with the same amount in the downward direction.

Arbitrary horizontal motions are also available — \h is quite analogous to \v, except that the default scale factor is ems instead of line spaces. As an example,

```
\h'-0.1i'
```

causes a backwards motion of a tenth of an inch. As a practical matter, consider printing the mathematical symbol '>>'. The default spacing is too wide, so **eqn** replaces this by

```
>\h'-0.3m'>
```

to produce >>.

Frequently \h is used with the 'width function' \w to generate motions equal to the width of some character string. The construction

```
\w'thing'
```

is a number equal to the width of 'thing' in machine units (1/432 inch). All **troff** computations are ultimately done in these units. To move horizontally the width of an 'x', we can say

```
\h'\w'x'u'
```

As we mentioned above, the default scale factor for all horizontal dimensions is **m**, ems, so here we must have the **u** for machine units, or the motion produced will be far too large. **troff** is quite happy with the nested quotes, by the way, so long as you don't leave any out.

As a live example of this kind of construction, all of the command names in the text, like .sp, were done by overstriking with a slight offset. The commands for .sp are

```
.sp\h'-\w'.sp'u'\h'1u'.sp
```

That is, put out '.sp', move left by the width of '.sp', move right 1 unit, and print '.sp' again. (Of course there is a way to avoid typing that much input for each command name, which we will discuss in Section 11.)

There are also several special-purpose **troff** commands for local motion. We have already seen \0, which is an unpaddable white space of the same width as a digit. 'Unpaddable' means that it will never be widened or split across a line by line justification and filling. There is also \(blank), which is an unpaddable character the width of a space, \|, which is half that width, \^, which is one quarter of the width of a space, and \&, which has zero width. (This last one is useful, for example, in entering a text line which would otherwise begin with a '.'.)

The command \o, used like

```
\o'set of characters'
```

causes (up to 9) characters to be overstruck, centered on the widest. This is nice for accents, as in

```
syst\o"e\(ga"me t\o"e\(aa"l\o"e\(aa"phonique
```

which makes

système téléphonique

The accents are \(ga and \(aa, or \` and \'; remember that each is just one character to **troff**.

You can make your own overstrikes with another special convention, \z, the zero-motion command. \zx suppresses the normal horizontal motion after printing the single character x, so another character can be laid on top of it. Although sizes can be changed within \o, it centers the characters on the widest, and there can be no horizontal or vertical motions, so \z may be the only way to get what you want:

is produced by

```
.sp 2
\s8\z\(sq\s14\z\(sq\s22\z\(sq\s36\(sq
```

The .sp is needed to leave room for the result.

As another example, an extra-heavy semicolon that looks like

; instead of ; or ;

can be constructed with a big comma and a big period above it:

```
\s+6\z,\v'-0.25m'.\v'0.25m'\s0
```

'0.25m' is an empirical constant.

A more ornate overstrike is given by the bracketing function \b, which piles up characters vertically, centered on the current baseline. Thus we can get big brackets, constructing them with piled-up smaller pieces:

by typing in only this:

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

troff also provides a convenient facility for drawing horizontal and vertical lines of arbitrary length with arbitrary characters. \l'1i' draws a line one inch long, like this: _____. The length can be followed by the character to use if the _ isn't appropriate; \l'0.5i.' draws a half-inch line of dots: ..... The construction \L is entirely analogous, except that it draws a vertical line instead of horizontal.

7. Strings

Obviously if a paper contains a large number of occurrences of an acute accent over a letter 'e', typing \o"e\'" for each é would be a

great nuisance.

Fortunately, **troff** provides a way in which you can store an arbitrary collection of text in a 'string', and thereafter use the string name as a shorthand for its contents. Strings are one of several **troff** mechanisms whose judicious use lets you type a document with less effort and organize it so that extensive format changes can be made with few editing changes.

A reference to a string is replaced by whatever text the string was defined as. Strings are defined with the command .ds. The line

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

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

String names may be either one or two characters long, and are referred to by *x for one character names or *(xy for two character names. Thus to get téléphone, given the definition of the string e as above, we can say t*el*ephone.

If a string must begin with blanks, define it as

```
.ds xx "     text
```

The double quote signals the beginning of the definition. There is no trailing quote; the end of the line terminates the string.

A string may actually be several lines long; if **troff** encounters a \ at the end of *any* line, it is thrown away and the next line added to the current one. So you can make a long string simply by ending each line but the last with a backslash:

```
.ds xx this \
is a very \
long string
```

Strings may be defined in terms of other strings, or even in terms of themselves; we will discuss some of these possibilities later.

8. Introduction to Macros

Before we can go much further in **troff**, we need to learn a bit about the macro facility. In its simplest form, a macro is just a shorthand notation quite similar to a string. Suppose we want every paragraph to start in exactly the same way — with a space and a temporary indent of two ems:

```
.sp
.ti +2m
```

Then to save typing, we would like to collapse these into one shorthand line, a **troff** 'command' like

```
.PP
```

that would be treated by **troff** exactly as

```
.sp
.ti +2m
```

.PP is called a *macro*. The way we tell **troff** what .PP means is to *define* it with the .de command:

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

The first line names the macro (we used '.PP' for 'paragraph', and upper case so it wouldn't conflict with any name that **troff** might already know about). The last line .. marks the end of the definition. In between is the text, which is simply inserted whenever **troff** sees the 'command' or macro call

```
.PP
```

A macro can contain any mixture of text and formatting commands.

The definition of .PP has to precede its first use; undefined macros are simply ignored. Names are restricted to one or two characters.

Using macros for commonly occurring sequences of commands is critically important. Not only does it save typing, but it makes later changes much easier. Suppose we decide that the paragraph indent is too small, the vertical space is much too big, and roman font should be forced. Instead of changing the whole document, we need only change the definition of .PP to something like

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

and the change takes effect everywhere we used .PP.

\" is a **troff** command that causes the rest of the line to be ignored. We use it here to add comments to the macro definition (a wise idea once definitions get complicated).

As another example of macros, consider these two which start and end a block of offset, unfilled text, like most of the examples in this paper:

```
.de BS          \" start indented block
.sp
.nf
.in +0.3i
..
.de BE          \" end indented block
.sp
.fi
.in -0.3i
..
```

Now we can surround text like

> Copy to
> John Doe
> Richard Roberts
> Stanley Smith

by the commands .BS and .BE, and it will come out as it did above. Notice that we indented by .in +0.3i instead of .in 0.3i. This way we can nest our uses of .BS and BE to get blocks within blocks.

If later on we decide that the indent should be 0.5i, then it is only necessary to change the definitions of .BS and .BE, not the whole paper.

9. Titles, Pages and Numbering

This is an area where things get tougher, because nothing is done for you automatically. Of necessity, some of this section is a cookbook, to be copied literally until you get some experience.

Suppose you want a title at the top of each page, saying just

"----left top center top right top----

In **roff**, one can say

```
.he 'left top'center top'right top'
.fo 'left bottom'center bottom'right bottom'
```

to get headers and footers automatically on every page. Alas, this doesn't work in **troff**, a serious hardship for the novice. Instead you have to do a lot of specification.

You have to say what the actual title is (easy); when to print it (easy enough); and what to do at and around the title line (harder). Taking these in reverse order, first we define a macro .NP (for 'new page') to process titles and the like at the end of one page and the beginning of the next:

```
.de NP
'bp
'sp 0.5i
.tl 'left top'center top'right top'
'sp 0.3i
..
```

To make sure we're at the top of a page, we issue a 'begin page' command 'bp, which causes a skip to top-of-page (we'll explain the ' shortly). Then we space down half an inch, print the title (the use of .tl should be self explanatory; later we will discuss parameterizing the titles), space another 0.3 inches, and we're done.

To ask for .NP at the bottom of each page, we have to say something like 'when the text is within an inch of the bottom of the page, start the processing for a new page.' This is done with a 'when' command .wh:

```
.wh -1i NP
```

(No '.' is used before NP; this is simply the name of a macro, not a macro call.) The minus sign means 'measure up from the bottom of the page', so '-1i' means 'one inch from the bottom'.

The .wh command appears in the input outside the definition of .NP; typically the input would be

```
.de NP
...
..
.wh -1i NP
```

Now what happens? As text is actually being output, **troff** keeps track of its vertical position on the page, and after a line is printed within one inch from the bottom, the .NP macro is activated. (In the jargon, the .wh command sets a *trap* at the specified place, which is 'sprung' when that point is passed.) .NP causes a skip to the top of the next page (that's what the 'bp was for), then prints the title with the appropriate margins.

Why 'bp and 'sp instead of .bp and .sp? The answer is that .sp and .bp, like several other commands, cause a *break* to take place. That is, all the input text collected but not yet printed is flushed out as soon as possible, and the next input line is guaranteed to start a new line of output. If we had used .sp or .bp in the .NP macro, this would cause a break in the middle of the current output line when a new page is started. The effect would be to print the left-over part of that line at the top of the page, followed by the next input line on a new output line. This is *not* what we want. Using ' instead of . for a command tells **troff** that no break is to take place — the output line currently being filled should *not* be forced out before the space or new page.

The list of commands that cause a break is short and natural:

```
.bp  .br  .ce  .fi  .nf  .sp  .in  .ti
```

All others cause *no* break, regardless of whether

you use a . or a '. If you really need a break, add a .br command at the appropriate place.

One other thing to beware of — if you're changing fonts or point sizes a lot, you may find that if you cross a page boundary in an unexpected font or size, your titles come out in that size and font instead of what you intended. Furthermore, the length of a title is independent of the current line length, so titles will come out at the default length of 6.5 inches unless you change it, which is done with the .lt command.

There are several ways to fix the problems of point sizes and fonts in titles. For the simplest applications, we can change .NP to set the proper size and font for the title, then restore the previous values, like this:

```
.de NP
'bp
'sp 0.5i
.ft R           \" set title font to roman
.ps 10          \" and size to 10 point
.lt 6i          \" and length to 6 inches
.tl 'left'center'right'
.ps             \" revert to previous size
.ft P           \" and to previous font
'sp 0.3i
..
```

This version of .NP does *not* work if the fields in the .tl command contain size or font changes. To cope with that requires **troff**'s 'environment' mechanism, which we will discuss in Section 13.

To get a footer at the bottom of a page, you can modify .NP so it does some processing before the 'bp command, or split the job into a footer macro invoked at the bottom margin and a header macro invoked at the top of the page. These variations are left as exercises.

Output page numbers are computed automatically as each page is produced (starting at 1), but no numbers are printed unless you ask for them explicitly. To get page numbers printed, include the character % in the .tl line at the position where you want the number to appear. For example

```
.tl "- % -"
```

centers the page number inside hyphens, as on this page. You can set the page number at any time with either .bp n, which immediately starts a new page numbered n, or with .pn n, which sets the page number for the next page but doesn't cause a skip to the new page. Again, .bp +n sets the page number to n more than its current value; .bp means .bp +1.

10. Number Registers and Arithmetic

troff has a facility for doing arithmetic, and for defining and using variables with numeric values, called *number registers*. Number registers, like strings and macros, can be useful in setting up a document so it is easy to change later. And of course they serve for any sort of arithmetic computation.

Like strings, number registers have one or two character names. They are set by the .nr command, and are referenced anywhere by \nx (one character name) or \n(xy (two character name).

There are quite a few pre-defined number registers maintained by **troff**, among them % for the current page number; nl for the current vertical position on the page; dy, mo and yr for the current day, month and year; and .s and .f for the current size and font. (The font is a number from 1 to 4.) Any of these can be used in computations like any other register, but some, like .s and .f, cannot be changed with .nr.

As an example of the use of number registers, in the −ms macro package [4], most significant parameters are defined in terms of the values of a handful of number registers. These include the point size for text, the vertical spacing, and the line and title lengths. To set the point size and vertical spacing for the following paragraphs, for example, a user may say

```
.nr PS 9
.nr VS 11
```

The paragraph macro .PP is defined (roughly) as follows:

```
.de PP
.ps \\n(PS      \" reset size
.vs \\n(VSp     \" spacing
.ft R           \" font
.sp 0.5v        \" half a line
.ti +3m
..
```

This sets the font to Roman and the point size and line spacing to whatever values are stored in the number registers PS and VS.

Why are there two backslashes? This is the eternal problem of how to quote a quote. When **troff** originally reads the macro definition, it peels off one backslash to see what's coming next. To ensure that another is left in the definition when the macro is *used*, we have to put in two backslashes in the definition. If only one backslash is used, point size and vertical spacing will be frozen at the time the macro is defined, not when it is used.

Protecting by an extra layer of backslashes

is only needed for \n, *, \$ (which we haven't come to yet), and \ itself. Things like \s, \f, \h, \v, and so on do not need an extra backslash, since they are converted by **troff** to an internal code immediately upon being seen.

Arithmetic expressions can appear anywhere that a number is expected. As a trivial example,

```
.nr PS \\n(PS-2
```

decrements PS by 2. Expressions can use the arithmetic operators +, −, *, /, % (mod), the relational operators >, >=, <, <=, ==, and != (not equal), and parentheses.

Although the arithmetic we have done so far has been straightforward, more complicated things are somewhat tricky. First, number registers hold only integers. **troff** arithmetic uses truncating integer division, just like Fortran. Second, in the absence of parentheses, evaluation is done left-to-right without any operator precedence (including relational operators). Thus

```
7*-4+3/13
```

becomes '−1'. Number registers can occur anywhere in an expression, and so can scale indicators like p, i, m, and so on (but no spaces). Although integer division causes truncation, each number and its scale indicator is converted to machine units (1/432 inch) before any arithmetic is done, so 1i/2u evaluates to 0.5i correctly.

The scale indicator u often has to appear when you wouldn't expect it — in particular, when arithmetic is being done in a context that implies horizontal or vertical dimensions. For example,

```
.ll 7/2i
```

would seem obvious enough — 3½ inches. Sorry. Remember that the default units for horizontal parameters like .ll are ems. That's really '7 ems / 2 inches', and when translated into machine units, it becomes zero. How about

```
.ll 7i/2
```

Sorry, still no good — the '2' is '2 ems', so '7i/2' is small, although not zero. You *must* use

```
.ll 7i/2u
```

So again, a safe rule is to attach a scale indicator to every number, even constants.

For arithmetic done within a .nr command, there is no implication of horizontal or vertical dimension, so the default units are 'units', and 7i/2 and 7i/2u mean the same thing. Thus

```
.nr ll 7i/2
.ll \\n(llu
```

does just what you want, so long as you don't forget the u on the .ll command.

11. Macros with arguments

The next step is to define macros that can change from one use to the next according to parameters supplied as arguments. To make this work, we need two things: first, when we define the macro, we have to indicate that some parts of it will be provided as arguments when the macro is called. Then when the macro is called we have to provide actual arguments to be plugged into the definition.

Let us illustrate by defining a macro .SM that will print its argument two points smaller than the surrounding text. That is, the macro call

```
.SM TROFF
```

will produce TROFF.

The definition of .SM is

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

Within a macro definition, the symbol \\$n refers to the nth argument that the macro was called with. Thus \\$1 is the string to be placed in a smaller point size when .SM is called.

As a slightly more complicated version, the following definition of .SM permits optional second and third arguments that will be printed in the normal size:

```
.de SM
\\$3\s-2\\$1\s+2\\$2
..
```

Arguments not provided when the macro is called are treated as empty, so

```
.SM TROFF ),
```

produces TROFF), while

```
.SM TROFF ). (
```

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

By the way, the number of arguments that a macro was called with is available in number register .$.

The following macro .BD is the one used to make the 'bold roman' we have been using for **troff** command names in text. It combines horizontal motions, width computations, and argument rearrangement.

```
.de BD
\&\\$3\fI\\$1\h'-\w'\\$1'u+1u'\\$1\fP\\$2
..
```

The \h and \w commands need no extra backslash, as we discussed above. The \& is there in case the argument begins with a period.

Two backslashes are needed with the \\$n commands, though, to protect one of them when the macro is being defined. Perhaps a second example will make this clearer. Consider a macro called .SH which produces section headings rather like those in this paper, with the sections numbered automatically, and the title in bold in a smaller size. The use is

```
.SH "Section title ..."
```

(If the argument to a macro is to contain blanks, then it must be *surrounded* by double quotes, unlike a string, where only one leading quote is permitted.)

Here is the definition of the .SH macro:

```
.nr SH 0        \" initialize section number
.de SH
.sp 0.3i
.ft B
.nr SH \\n(SH+1    \" increment number
.ps \\n(PS-1       \" decrease PS
\\n(SH. \\$1       \" number. title
.ps \\n(PS         \" restore PS
.sp 0.3i
.ft R
..
```

The section number is kept in number register SH, which is incremented each time just before it is used. (A number register may have the same name as a macro without conflict but a string may not.)

We used \\n(SH instead of \n(SH and \\n(PS instead of \n(PS. If we had used \n(SH, we would get the value of the register at the time the macro was *defined*, not at the time it was *used*. If that's what you want, fine, but not here. Similarly, by using \\n(PS, we get the point size at the time the macro is called.

As an example that does not involve numbers, recall our .NP macro which had a

```
.tl 'left'center'right'
```

We could make these into parameters by using instead

```
.tl '\\*(LT'\\*(CT'\\*(RT'
```

so the title comes from three strings called LT, CT and RT. If these are empty, then the title will be a blank line. Normally CT would be set with something like

```
.ds CT - % -
```

to give just the page number between hyphens (as on the top of this page), but a user could supply private definitions for any of the strings.

12. Conditionals

Suppose we want the .SH macro to leave two extra inches of space just before section 1, but nowhere else. The cleanest way to do that is to test inside the .SH macro whether the section number is 1, and add some space if it is. The .if command provides the conditional test that we can add just before the heading line is output:

```
.if \\n(SH=1 .sp 2i        \" first section only
```

The condition after the .if can be any arithmetic or logical expression. If the condition is logically true, or arithmetically greater than zero, the rest of the line is treated as if it were text — here a command. If the condition is false, or zero or negative, the rest of the line is skipped.

It is possible to do more than one command if a condition is true. Suppose several operations are to be done before section 1. One possibility is to define a macro .S1 and invoke it if we are about to do section 1 (as determined by an .if).

```
.de S1
--- processing for section 1 ---
..
.de SH
...
.if \\n(SH=1 .S1
...
..
```

An alternate way is to use the extended form of the .if, like this:

```
.if \\n(SH=1 \{--- processing
for section 1 ----\}
```

The braces \{ and \} must occur in the positions shown or you will get unexpected extra lines in your output. **troff** also provides an 'if-else' construction, which we will not go into here.

A condition can be negated by preceding it with !; we get the same effect as above (but less clearly) by using

```
.if !\\n(SH>1 .S1
```

There are a handful of other conditions that can be tested with .if. For example, is the current page even or odd?

```
.if e .tl "even page title"
.if o .tl "odd page title"
```

gives facing pages different titles when used inside an appropriate new page macro.

Two other conditions are t and n, which tell you whether the formatter is **troff** or **nroff**.

```
.if t troff stuff ...
.if n nroff stuff ...
```

Finally, string comparisons may be made in an .if:

```
.if 'string1'string2' stuff
```

does 'stuff' if *string1* is the same as *string2*. The character separating the strings can be anything reasonable that is not contained in either string. The strings themselves can reference strings with *, arguments with \$, and so on.

13. Environments

As we mentioned, there is a potential problem when going across a page boundary: parameters like size and font for a page title may well be different from those in effect in the text when the page boundary occurs. **troff** provides a very general way to deal with this and similar situations. There are three 'environments', each of which has independently settable versions of many of the parameters associated with processing, including size, font, line and title lengths, fill/nofill mode, tab stops, and even partially collected lines. Thus the titling problem may be readily solved by processing the main text in one environment and titles in a separate one with its own suitable parameters.

The command .ev n shifts to environment n; n must be 0, 1 or 2. The command .ev with no argument returns to the previous environment. Environment names are maintained in a stack, so calls for different environments may be nested and unwound consistently.

Suppose we say that the main text is processed in environment 0, which is where **troff** begins by default. Then we can modify the new page macro .NP to process titles in environment 1 like this:

```
.de NP
.ev 1           \" shift to new environment
.lt 6i          \" set parameters here
.ft R
.ps 10
... any other processing ...
.ev             \" return to previous environment
..
```

It is also possible to initialize the parameters for an environment outside the .NP macro, but the version shown keeps all the processing in one place and is thus easier to understand and change.

14. Diversions

There are numerous occasions in page layout when it is necessary to store some text for a period of time without actually printing it. Footnotes are the most obvious example: the text of the footnote usually appears in the input well before the place on the page where it is to be printed is reached. In fact, the place where it is output normally depends on how big it is, which implies that there must be a way to process the footnote at least enough to decide its size without printing it.

troff provides a mechanism called a diversion for doing this processing. Any part of the output may be diverted into a macro instead of being printed, and then at some convenient time the macro may be put back into the input.

The command .di xy begins a diversion — all subsequent output is collected into the macro xy until the command .di with no arguments is encountered. This terminates the diversion. The processed text is available at any time thereafter, simply by giving the command

```
.xy
```

The vertical size of the last finished diversion is contained in the built-in number register dn.

As a simple example, suppose we want to implement a 'keep-release' operation, so that text between the commands .KS and .KE will not be split across a page boundary (as for a figure or table). Clearly, when a .KS is encountered, we have to begin diverting the output so we can find out how big it is. Then when a .KE is seen, we decide whether the diverted text will fit on the current page, and print it either there if it fits, or at the top of the next page if it doesn't. So:

```
.de KS      \" start keep
.br         \" start fresh line
.ev 1       \" collect in new environment
.fi         \" make it filled text
.di XX      \" collect in XX
..

.de KE      \" end keep
.br         \" get last partial line
.di         \" end diversion
.if \\n(dn>=\\n(.t .bp   \" bp if doesn't fit
.nf         \" bring it back in no-fill
.XX         \" text
.ev         \" return to normal environment
..
```

Recall that number register nl is the current

position on the output page. Since output was being diverted, this remains at its value when the diversion started. dn is the amount of text in the diversion; .t (another built-in register) is the distance to the next trap, which we assume is at the bottom margin of the page. If the diversion is large enough to go past the trap, the .if is satisfied, and a .bp is issued. In either case, the diverted output is then brought back with .XX. It is essential to bring it back in no-fill mode so **troff** will do no further processing on it.

This is not the most general keep-release, nor is it robust in the face of all conceivable inputs, but it would require more space than we have here to write it in full generality. This section is not intended to teach everything about diversions, but to sketch out enough that you can read existing macro packages with some comprehension.

Acknowledgements

I am deeply indebted to J. F. Ossanna, the author of **troff**, for his repeated patient explanations of fine points, and for his continuing willingness to adapt **troff** to make other uses easier. I am also grateful to Jim Blinn, Ted Dolotta, Doug McIlroy, Mike Lesk and Joel Sturman for helpful comments on this paper.

References

[1] J. F. Ossanna, *NROFF/TROFF* User's Manual, Bell Laboratories Computing Science Technical Report 54, 1976.

[2] B. W. Kernighan, *A System for Typesetting Mathematics — User's Guide (Second Edition)*, Bell Laboratories Computing Science Technical Report 17, 1977.

[3] M. E. Lesk, *TBL — A Program to Format Tables*, Bell Laboratories Computing Science Technical Report 49, 1976.

[4] M. E. Lesk, *Typing Documents on UNIX*, Bell Laboratories, 1978.

[5] J. R. Mashey and D. W. Smith, *PWB/MM — Programmer's Workbench Memorandum Macros*, Bell Laboratories internal memorandum.

Appendix A: Phototypesetter Character Set

These characters exist in roman, italic, and bold. To get the one on the left, type the four-character name on the right.

ff	\(ff	fi	\(fi	fl	\(fl	ffi	\(Fi	ffl	\(Fl
_	\(ru	—	\(em	¼	\(14	½	\(12	¾	\(34
©	\(co	°	\(de	†	\(dg	'	\(fm	¢	\(ct
®	\(rg	•	\(bu	□	\(sq	-	\(hy		

(In bold, \(sq is ■.)

The following are special-font characters:

+	\(pl	−	\(mi	×	\(mu	÷	\(di
=	\(eq	≡	\(-=	≥	\(>=	≤	\(<=
≠	\(!=	±	\(+-	¬	\(no	/	\(sl
~	\(ap	≅	\(~=	∝	\(pt	∇	\(gr
→	\(->	←	\(<-	↑	\(ua	↓	\(da
∫	\(is	∂	\(pd	∞	\(if	√	\(sr
⊂	\(sb	⊃	\(sp	∪	\(cu	∩	\(ca
⊆	\(ib	⊇	\(ip	∈	\(mo	∅	\(es
´	\(aa	`	\(ga	○	\(ci	Ⓐ	\(bs
§	\(sc	‡	\(dd	▬	\(lh	☛	\(rh
⌠	\(lt	⌡	\(rt	⎡	\(lc	⎤	\(rc
⎮	\(lb	⎮	\(rb	⎣	\(lf	⎦	\(rf
⎨	\(lk	⎬	\(rk	⎪	\(bv	ç	\(ts
|	\(br	|	\(or	_	\(ul	‾	\(rn
•	\(bu						

These four characters also have two-character names. The ´ is the apostrophe on terminals; the ` is the other quote mark.

´	\'	`	\`	–	\-	_	_

These characters exist only on the special font, but they do not have four-character names:

" { } < > ~ ^ \ # @

For greek, precede the roman letter by \(* to get the corresponding greek; for example, \(*a is α.

a b g d e z y h i k l m n c o p r s t u f x q w
$\alpha\ \beta\ \gamma\ \delta\ \epsilon\ \zeta\ \eta\ \theta\ \iota\ \kappa\ \lambda\ \mu\ \nu\ \xi\ o\ \pi\ \rho\ \sigma\ \tau\ \upsilon\ \phi\ \chi\ \psi\ \omega$

A B G D E Z Y H I K L M N C O P R S T U F X Q W
$A\ B\ \Gamma\ \Delta\ E\ Z\ H\ \Theta\ I\ K\ \Lambda\ M\ N\ \Xi\ O\ \Pi\ P\ \Sigma\ T\ Y\ \Phi\ X\ \Psi\ \Omega$

NROFF/TROFF User's Manual

Joseph F. Ossanna

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

NROFF and TROFF are text processors under the PDP-11 UNIX Time-Sharing System[1] that format text for typewriter-like terminals and for a Graphic Systems phototypesetter, respectively. They accept lines of text interspersed with lines of format control information and format the text into a printable, paginated document having a user-designed style. NROFF and TROFF offer unusual freedom in document styling, including: arbitrary style headers and footers; arbitrary style footnotes; multiple automatic sequence numbering for paragraphs, sections, etc; multiple column output; dynamic font and point-size control; arbitrary horizontal and vertical local motions at any point; and a family of automatic overstriking, bracket construction, and line drawing functions.

NROFF and TROFF are highly compatible with each other and it is almost always possible to prepare input acceptable to both. Conditional input is provided that enables the user to embed input expressly destined for either program. NROFF can prepare output directly for a variety of terminal types and is capable of utilizing the full resolution of each terminal.

Usage

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

nroff *options files* (or **troff** *options files*)

where *options* represents any of a number of option arguments and *files* represents the list of files containing the document to be formatted. An argument consisting of a single minus (−) is taken to be a file name corresponding to the standard input. If no file names are given input is taken from the standard input. The options, which may appear in any order so long as they appear before the files, are:

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

NROFF Only

−T*name* Specifies the name of the output terminal type. Currently defined names are 37 for the (default) Model 37 Teletype®, **tn300** for the GE TermiNet 300 (or any terminal without half-line capabilities), **300S** for the DASI-300S, **300** for the DASI-300, and **450** for the DASI-450 (Diablo Hyterm).

−e Produce equally-spaced words in adjusted lines, using full terminal resolution.

TROFF Only

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

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

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

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

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

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

−g Prepare output for the Murray Hill Computation Center phototypesetter and direct it to the standard output.

Each option is invoked as a separate argument; for example,

nroff **−o*4,8−10*** **−T*300S*** **−m*abc*** ***file1*** ***file2***

requests formatting of pages 4, 8, 9, and 10 of a document contained in the files named *file1* and *file2*, specifies the output terminal as a DASI-300S, and invokes the macro package *abc*.

Various pre- and post-processors are available for use with NROFF and TROFF. These include the equation preprocessors NEQN and EQN[2] (for NROFF and TROFF respectively), and the table construction preprocessor TBL[3]. A reverse-line postprocessor COL[4] is available for multiple-column NROFF output on terminals without reverse-line ability; COL expects the Model 37 Teletype escape sequences that NROFF produces by default. TK[4] is a 37 Teletype simulator postprocessor for printing NROFF output on a Tektronix 4014. TCAT[4] is phototypesetter-simulator postprocessor for TROFF that produces an approximation of phototypesetter output on a Tektronix 4014. For example, in

tbl ***files*** | **eqn** | **troff** **−t** ***options*** | **tcat**

the first | indicates the piping of TBL's output to EQN's input; the second the piping of EQN's output to TROFF's input; and the third indicates the piping of TROFF's output to TCAT. GCAT[4] can be used to send TROFF (**−g**) output to the Murray Hill Computation Center.

The remainder of this manual consists of: a Summary and Index; a Reference Manual keyed to the index; and a set of Tutorial Examples. Another tutorial is [5].

Joseph F. Ossanna

References

[1] K. Thompson, D. M. Ritchie, *UNIX Programmer's Manual*, Sixth Edition (May 1975).

[2] B. W. Kernighan, L. L. Cherry, *Typesetting Mathematics — User's Guide (Second Edition)*, Bell Laboratories internal memorandum.

[3] M. E. Lesk, *Tbl — A Program to Format Tables*, Bell Laboratories internal memorandum.

[4] Internal on-line documentation, on UNIX.

[5] B. W. Kernighan, *A TROFF Tutorial*, Bell Laboratories internal memorandum.

SUMMARY AND INDEX

Request Form	*Initial Value**	*If No Argument*	*Notes#*	*Explanation*
1. General Explanation				
2. Font and Character Size Control				
.ps ±*N*	10 point	previous	E	Point size; also \s±*N*.†
.ss *N*	12/36 em	ignored	E	Space-character size set to *N*/36 em.†
.cs *FNM*	off	-	P	Constant character space (width) mode (font *F*).†
.bd *F N*	off	-	P	Embolden font *F* by *N*−1 units.†
.bd S *F N*	off	-	P	Embolden Special Font when current font is *F*.†
.ft *F*	Roman	previous	E	Change to font *F* = *x*, *xx*, or 1-4. Also \f*x*, \f(*xx*, \f*N*.
.fp *N F*	R,I,B,S	ignored	-	Font named *F* mounted on physical position $1 \leqslant N \leqslant 4$.
3. Page Control				
.pl ±*N*	11 in	11 in	v	Page length.
.bp ±*N*	*N*=1	-	B‡,v	Eject current page; next page number *N*.
.pn ±*N*	*N*=1	ignored	-	Next page number *N*.
.po ±*N*	0; 26/27 in	previous	v	Page offset.
.ne *N*	-	*N*=1 *V*	D,v	Need *N* vertical space (*V* = vertical spacing).
.mk *R*	none	internal	D	Mark current vertical place in register *R*.
.rt ±*N*	none	internal	D,v	Return *(upward only)* to marked vertical place.
4. Text Filling, Adjusting, and Centering				
.br	-	-	B	Break.
.fi	fill	-	B,E	Fill output lines.
.nf	fill	-	B,E	No filling or adjusting of output lines.
.ad *c*	adj,both	adjust	E	Adjust output lines with mode *c*.
.na	adjust	-	E	No output line adjusting.
.ce *N*	off	*N*=1	B,E	Center following *N* input text lines.
5. Vertical Spacing				
.vs *N*	1/6in;12pts	previous	E,p	Vertical base line spacing (*V*).
.ls *N*	*N*=1	previous	E	Output *N*−1 *V*s after each text output line.
.sp *N*	-	*N*=1 *V*	B,v	Space vertical distance *N in either direction*.
.sv *N*	-	*N*=1 *V*	v	Save vertical distance *N*.
.os	-	-	-	Output saved vertical distance.
.ns	space	-	D	Turn no-space mode on.
.rs	-	-	D	Restore spacing; turn no-space mode off.
6. Line Length and Indenting				
.ll ±*N*	6.5 in	previous	E,m	Line length.
.in ±*N*	*N*=0	previous	B,E,m	Indent.
.ti ±*N*	-	ignored	B,E,m	Temporary indent.
7. Macros, Strings, Diversion, and Position Traps				
.de *xx yy*	-	*.yy*=..	-	Define or redefine macro *xx*; end at call of *.yy*.
.am *xx yy*	-	*.yy*=..	-	Append to a macro.
.ds *xx string*	-	ignored	-	Define a string *xx* containing *string*.
.as *xx string*	-	ignored	-	Append *string* to string *xx*.

*Values separated by ";" are for NROFF and TROFF respectively.

#Notes are explained at the end of this Summary and Index

†No effect in NROFF.

‡The use of "'" as control character (instead of ".") suppresses the break function.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.rm *xx*	-	ignored	-	Remove request, macro, or string.
.rn *xx yy*	-	ignored	-	Rename request, macro, or string *xx* to *yy*.
.di *xx*	-	end	D	Divert output to macro *xx*.
.da *xx*	-	end	D	Divert and append to *xx*.
.wh *N xx*	-	-	v	Set location trap; negative is w.r.t. page bottom.
.ch *xx N*	-	-	v	Change trap location.
.dt *N xx*	-	off	D,v	Set a diversion trap.
.it *N xx*	-	off	E	Set an input-line count trap.
.em *xx*	none	none	-	End macro is *xx*.
8. Number Registers				
.nr *R ±N M*		-	u	Define and set number register *R*; auto-increment by *M*.
.af *R c*	arabic	-	-	Assign format to register *R* (*c*=1, i, I, a, A).
.rr *R*	-	-	-	Remove register *R*.
9. Tabs, Leaders, and Fields				
.ta *Nt* ...	0.8; 0.5in	none	E,m	Tab settings; *left* type, unless *t*=R(right), C(centered).
.tc *c*	none	none	E	Tab repetition character.
.lc *c*	.	none	E	Leader repetition character.
.fc *a b*	off	off	-	Set field delimiter *a* and pad character *b*.
10. Input and Output Conventions and Character Translations				
.ec *c*	\	\	-	Set escape character.
.eo	on	-	-	Turn off escape character mechanism.
.lg *N*	-; on	on	-	Ligature mode on if *N*>0.
.ul *N*	off	*N*=1	E	Underline (italicize in TROFF) *N* input lines.
.cu *N*	off	*N*=1	E	Continuous underline in NROFF; like ul in TROFF.
.uf *F*	Italic	Italic	-	Underline font set to *F* (to be switched to by ul).
.cc *c*	.	.	E	Set control character to *c*.
.c2 *c*	'	'	E	Set nobreak control character to *c*.
.tr *abcd*....	none	-	O	Translate *a* to *b*, etc. on output.
11. Local Horizontal and Vertical Motions, and the Width Function				
12. Overstrike, Bracket, Line-drawing, and Zero-width Functions				
13. Hyphenation.				
.nh	hyphenate	-	E	No hyphenation.
.hy *N*	hyphenate	hyphenate	E	Hyphenate; *N* = mode.
.hc *c*	\%	\%	E	Hyphenation indicator character *c*.
.hw *word1* ...		ignored	-	Exception words.
14. Three Part Titles.				
.tl *'left'center'right'*		-	-	Three part title.
.pc *c*	%	off	-	Page number character.
.lt *±N*	6.5 in	previous	E,m	Length of title.
15. Output Line Numbering.				
.nm *±N M S I*		off	E	Number mode on or off, set parameters.
.nn *N*	-	*N*=1	E	Do not number next *N* lines.
16. Conditional Acceptance of Input				
.if *c anything*		-	-	If condition *c* true, accept *anything* as input, for multi-line use \{*anything*\}.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.if !*c anything*		-	-	If condition *c* false, accept *anything.*
.if *N anything*		-	u	If expression $N > 0$, accept *anything.*
.if !*N anything*		-	u	If expression $N \leqslant 0$, accept *anything.*
.if *'string1'string2' anything*			-	If *string1* identical to *string2*, accept *anything.*
.if !*'string1'string2' anything*			-	If *string1* not identical to *string2*, accept *anything.*
.ie *c anything*		-	u	If portion of if-else; all above forms (like if).
.el *anything*		-	-	Else portion of if-else.
17. Environment Switching.				
.ev *N*	*N*=0	previous	-	Environment switched (*push down*).
18. Insertions from the Standard Input				
.rd *prompt*	-	*prompt*=BEL	-	Read insertion.
.ex	-	-	-	Exit from NROFF/TROFF.
19. Input/Output File Switching				
.so *filename*		-	-	Switch source file *(push down).*
.nx *filename*		end-of-file	-	Next file.
.pi *program*		-	-	Pipe output to *program* (NROFF only).
20. Miscellaneous				
.mc *c N*	-	off	E,m	Set margin character *c* and separation *N.*
.tm *string*	-	newline	-	Print *string* on terminal (UNIX standard message output).
.ig *yy*	-	*.yy*=..	-	Ignore till call of *yy.*
.pm *t*	-	all	-	Print macro names and sizes; if *t* present, print only total of sizes.
.fl	-	-	B	Flush output buffer.
21. Output and Error Messages				

Notes-

- B Request normally causes a break.
- D Mode or relevant parameters associated with current diversion level.
- E Relevant parameters are a part of the current environment.
- O Must stay in effect until logical output.
- P Mode must be still or again in effect at the time of physical output.

v,p,m,u Default scale indicator; if not specified, scale indicators are *ignored.*

Alphabetical Request and Section Number Cross Reference

Escape Sequences for Characters, Indicators, and Functions

Section Reference	*Escape Sequence*	*Meaning*
10.1	\\	\ (to prevent or delay the interpretation of \)
10.1	\e	Printable version of the *current* escape character.
2.1	\'	' (acute accent); equivalent to \(aa
2.1	\`	` (grave accent); equivalent to \(ga
2.1	\-	− Minus sign in the *current* font
7	\.	Period (dot) (see de)
11.1	\(space)	Unpaddable space-size space character
11.1	\0	Digit width space
11.1	\\|	1/6 em narrow space character (zero width in NROFF)
11.1	\^	1/12 em half-narrow space character (zero width in NROFF)
4.1	\&	Non-printing, zero width character
10.6	\!	Transparent line indicator
10.7	\"	Beginning of comment
7.3	\$*N*	Interpolate argument $1 \leqslant N \leqslant 9$
13	\%	Default optional hyphenation character
2.1	\(*xx*	Character named *xx*
7.1	*x, *(*xx*	Interpolate string *x* or *xx*
9.1	\a	Non-interpreted leader character
12.3	\b'*abc...*'	Bracket building function
4.2	\c	Interrupt text processing
11.1	\d	Forward (down) 1/2 em vertical motion (1/2 line in NROFF)
2.2	\f*x*,\f(*xx*,\f*N*	Change to font named *x* or *xx*, or position *N*
11.1	\h'*N*'	Local horizontal motion; move right *N (negative left)*
11.3	\k*x*	Mark horizontal *input* place in register *x*
12.4	\l'*Nc*'	Horizontal line drawing function (optionally with *c*)
12.4	\L'*Nc*'	Vertical line drawing function (optionally with *c*)
8	\n*x*,\n(*xx*	Interpolate number register *x* or *xx*
12.1	\o'*abc...*'	Overstrike characters *a*, *b*, *c*, ...
4.1	\p	Break and spread output line
11.1	\r	Reverse 1 em vertical motion (reverse line in NROFF)
2.3	\s*N*, \s±*N*	Point-size change function
9.1	\t	Non-interpreted horizontal tab
11.1	\u	Reverse (up) 1/2 em vertical motion (1/2 line in NROFF)
11.1	\v'*N*'	Local vertical motion; move down *N (negative up)*
11.2	\w'*string*'	Interpolate width of *string*
5.2	\x'*N*'	Extra line-space function *(negative before, positive after)*
12.2	\z*c*	Print *c* with zero width (without spacing)
16	\{	Begin conditional input
16	\}	End conditional input
10.7	\(newline)	Concealed (ignored) newline
-	*X*	*X*, any character *not* listed above

The escape sequences \\, \., \", \$, *, \a, \n, \t, and \(newline) are interpreted in *copy mode* (§7.2).

Predefined General Number Registers

Section Reference	*Register Name*	*Description*
3	**%**	Current page number.
11.2	**ct**	Character type (set by *width* function).
7.4	**dl**	Width (maximum) of last completed diversion.
7.4	**dn**	Height (vertical size) of last completed diversion.
-	**dw**	Current day of the week (1-7).
-	**dy**	Current day of the month (1-31).
11.3	**hp**	Current horizontal place on *input* line.
15	**ln**	Output line number.
-	**mo**	Current month (1-12).
4.1	**nl**	Vertical position of last printed text base-line.
11.2	**sb**	Depth of string below base line (generated by *width* function).
11.2	**st**	Height of string above base line (generated by *width* function).
-	**yr**	Last two digits of current year.

Predefined Read-Only Number Registers

Section Reference	*Register Name*	*Description*
7.3	**.$**	Number of arguments available at the current macro level.
-	**.A**	Set to 1 in TROFF, if **-a** option used; always 1 in NROFF.
11.1	**.H**	Available horizontal resolution in basic units.
-	**.T**	Set to 1 in NROFF, if **-T** option used; always 0 in TROFF.
11.1	**.V**	Available vertical resolution in basic units.
5.2	**.a**	Post-line extra line-space most recently utilized using **\x**'*N*'.
-	**.c**	Number of *lines* read from current input file.
7.4	**.d**	Current vertical place in current diversion; equal to **nl**, if no diversion.
2.2	**.f**	Current font as physical quadrant (1-4).
4	**.h**	Text base-line high-water mark on current page or diversion.
6	**.i**	Current indent.
6	**.l**	Current line length.
4	**.n**	Length of text portion on previous output line.
3	**.o**	Current page offset.
3	**.p**	Current page length.
2.3	**.s**	Current point size.
7.5	**.t**	Distance to the next trap.
4.1	**.u**	Equal to 1 in fill mode and 0 in nofill mode.
5.1	**.v**	Current vertical line spacing.
11.2	**.w**	Width of previous character.
-	**.x**	Reserved version-dependent register.
-	**.y**	Reserved version-dependent register.
7.4	**.z**	Name of current diversion.

REFERENCE MANUAL

1. General Explanation

1.1. Form of input. Input consists of *text lines*, which are destined to be printed, interspersed with *control lines*, which set parameters or otherwise control subsequent processing. Control lines begin with a *control character*—normally . (period) or ' (acute accent)—followed by a one or two character name that specifies a basic *request* or the substitution of a user-defined *macro* in place of the control line. The control character ' suppresses the *break* function—the forced output of a partially filled line—caused by certain requests. The control character may be separated from the request/macro name by white space (spaces and/or tabs) for esthetic reasons. Names must be followed by either space or newline. Control lines with unrecognized names are ignored.

Various special functions may be introduced anywhere in the input by means of an *escape* character, normally \. For example, the function \n*R* causes the interpolation of the contents of the *number register R* in place of the function; here *R* is either a single character name as in \n*x*, or left-parenthesis-introduced, two-character name as in \n(*xx*.

1.2. Formatter and device resolution. TROFF internally uses 432 units/inch, corresponding to the Graphic Systems phototypesetter which has a horizontal resolution of 1/432 inch and a vertical resolution of 1/144 inch. NROFF internally uses 240 units/inch, corresponding to the least common multiple of the horizontal and vertical resolutions of various typewriter-like output devices. TROFF rounds horizontal/vertical numerical parameter input to the actual horizontal/vertical resolution of the Graphic Systems typesetter. NROFF similarly rounds numerical input to the actual resolution of the output device indicated by the **-T** option (default Model 37 Teletype).

1.3. Numerical parameter input. Both NROFF and TROFF accept numerical input with the appended scale indicators shown in the following table, where *S* is the current type size in points, *V* is the current vertical line spacing in basic units, and *C* is a *nominal character width* in basic units.

Scale Indicator	Meaning	Number of basic units TROFF	NROFF
i	Inch	432	240
c	Centimeter	432×50/127	240×50/127
P	Pica = 1/6 inch	72	240/6
m	Em = *S* points	6×*S*	*C*
n	En = Em/2	3×*S*	*C, same as Em*
p	Point = 1/72 inch	6	240/72
u	Basic unit	1	1
v	Vertical line space	*V*	*V*
none	Default, see below		

In NROFF, *both* the em and the en are taken to be equal to the *C*, which is output-device dependent; common values are 1/10 and 1/12 inch. Actual character widths in NROFF need not be all the same and constructed characters such as -> (—) are often extra wide. The default scaling is ems for the horizontally-oriented requests and functions **ll**, **in**, **ti**, **ta**, **lt**, **po**, **mc**, \h, and \l; *V*s for the vertically-oriented requests and functions **pl**, **wh**, **ch**, **dt**, **sp**, **sv**, **ne**, **rt**, \v, \x, and \L; **p** for the **vs** request; and **u** for the requests **nr**, **if**, and **ie**. *All* other requests ignore any scale indicators. When a number register containing an already appropriately scaled number is interpolated to provide numerical input, the unit scale indicator **u** may need to be appended to prevent an additional inappropriate default scaling.

The number, *N*, may be specified in decimal-fraction form but the parameter finally stored is rounded to an integer number of basic units.

The *absolute position* indicator | may be prepended to a number *N* to generate the distance to the vertical or horizontal place *N*. For vertically-oriented requests and functions, |*N* becomes the distance in basic units from the current vertical place on the page or in a *diversion* (§7.4) to the the vertical place *N*. For *all* other requests and functions, |*N* becomes the distance from the current horizontal place on the *input* line to the horizontal place *N*. For example,

```
.sp |3.2c
```

will space *in the required direction* to 3.2 centimeters from the top of the page.

1.4. Numerical expressions. Wherever numerical input is expected an expression involving parentheses, the arithmetic operators +, −, /, *, % (mod), and the logical operators <, >, <=, >=, = (or ==), & (and), : (or) may be used. Except where controlled by parentheses, evaluation of expressions is left-to-right; there is no operator precedence. In the case of certain requests, an initial + or − is stripped and interpreted as an increment or decrement indicator respectively. In the presence of default scaling, the desired scale indicator must be attached to *every* number in an expression for which the desired and default scaling differ. For example, if the number register **x** contains 2 and the current point size is 10, then

```
.ll (4.25i+\nxP+3)/2u
```

will set the line length to 1/2 the sum of 4.25 inches + 2 picas + 30 points.

1.5. Notation. Numerical parameters are indicated in this manual in two ways. ±*N* means that the argument may take the forms *N*, +*N*, or −*N* and that the corresponding effect is to set the affected parameter to *N*, to increment it by *N*, or to decrement it by *N* respectively. Plain *N* means that an initial algebraic sign is *not* an increment indicator, but merely the sign of *N*. Generally, unreasonable numerical input is either ignored or truncated to a reasonable value. For example, most requests expect to set parameters to non-negative values; exceptions are **sp**, **wh**, **ch**, **nr**, and **if**. The requests **ps**, **ft**, **po**, **vs**, **ls**, **ll**, **in**, and **lt** restore the *previous* parameter value in the *absence* of an argument.

Single character arguments are indicated by single lower case letters and one/two character arguments are indicated by a pair of lower case letters. Character string arguments are indicated by multi-character mnemonics.

2. Font and Character Size Control

2.1. Character set. The TROFF character set consists of the Graphics Systems Commercial II character set plus a Special Mathematical Font character set—each having 102 characters. These character sets are shown in the attached Table I. All ASCII characters are included, with some on the Special Font. With three exceptions, the ASCII characters are input as themselves, and non-ASCII characters are input in the form \(*xx* where *xx* is a two-character name given in the attached Table II. The three ASCII exceptions are mapped as follows:

ASCII Input		Printed by TROFF	
Character	Name	Character	Name
'	acute accent	’	close quote
`	grave accent	‘	open quote
−	minus	-	hyphen

The characters ', `, and − may be input by \', \`, and \− respectively or by their names (Table II). The ASCII characters @, #, ", ', `, <, >, \, {, }, ~, ^, and _ exist only on the Special Font and are printed as a 1-em space if that Font is not mounted.

NROFF understands the entire TROFF character set, but can in general print only ASCII characters, additional characters as may be available on the output device, such characters as may be able to be constructed by overstriking or other combination, and those that can reasonably be mapped into other printable characters. The exact behavior is determined by a driving table prepared for each device. The

characters ', `, and _ print as themselves.

2.2. Fonts. The default mounted fonts are Times Roman (R), Times Italic (I), Times Bold (B), and the Special Mathematical Font (S) on physical typesetter positions 1, 2, 3, and 4 respectively. These fonts are used in this document. The *current* font, initially Roman, may be changed (among the mounted fonts) by use of the ft request, or by imbedding at any desired point either \f*x*, \f(*xx*, or \f*N* where *x* and *xx* are the name of a mounted font and *N* is a numerical font position. It is *not* necessary to change to the Special font; characters on that font are automatically handled. A request for a named but not-mounted font is *ignored*. TROFF can be informed that any particular font is mounted by use of the fp request. The list of known fonts is installation dependent. In the subsequent discussion of font-related requests, *F* represents either a one/two-character font name or the numerical font position, 1-4. The current font is available (as numerical position) in the read-only number register .f.

NROFF understands font control and normally underlines Italic characters (see §10.5).

2.3. Character size. Character point sizes available on the Graphic Systems typesetter are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36. This is a range of 1/12 inch to 1/2 inch. The ps request is used to change or restore the point size. Alternatively the point size may be changed between any two characters by imbedding a \s*N* at the desired point to set the size to *N*, or a \s±*N* ($1 \leqslant N \leqslant 9$) to increment/decrement the size by *N*; \s0 restores the *previous* size. Requested point size values that are between two valid sizes yield the larger of the two. The current size is available in the .s register. NROFF ignores type size control.

Request Form	*Initial Value*	*If No Argument*	*Notes**	*Explanation*
.ps ±*N*	10 point	previous	E	Point size set to ±*N*. Alternatively imbed \s*N* or \s±*N*. Any positive size value may be requested; if invalid, the next larger valid size will result, with a maximum of 36. A paired sequence +*N*, −*N* will work because the previous requested value is also remembered. Ignored in NROFF.
.ss *N*	12/36 em	ignored	E	Space-character size is set to *N*/36 ems. This size is the minimum word spacing in adjusted text. Ignored in NROFF.
.cs *FNM*	off	-	P	Constant character space (width) mode is set on for font *F* (if mounted); the width of every character will be taken to be *N*/36 ems. If *M* is absent, the em is that of the character's point size; if *M* is given, the em is *M*-points. All affected characters are centered in this space, including those with an actual width larger than this space. Special Font characters occurring while the current font is *F* are also so treated. If *N* is absent, the mode is turned off. The mode must be still or again in effect when the characters are physically printed. Ignored in NROFF.
.bd *F N*	off	-	P	The characters in font *F* will be artificially emboldened by printing each one twice, separated by *N*−1 basic units. A reasonable value for *N* is 3 when the character size is in the vicinity of 10 points. If *N* is missing the embolden mode is turned off. The column heads above were printed with .bd I 3. The mode must be still or again in effect when the characters are physically printed. Ignored in NROFF.

*Notes are explained at the end of the Summary and Index above.

.bd S *F N*	off	-	P	The characters in the Special Font will be emboldened whenever the current font is *F*. This manual was printed with **.bd S B** 3. The mode must be still or again in effect when the characters are physically printed.
.ft *F*	Roman	previous	E	Font changed to *F*. Alternatively, imbed **\f***F*. The font name **P** is reserved to mean the previous font.
.fp *N F*	R,I,B,S	ignored	-	Font position. This is a statement that a font named *F* is mounted on position *N* (1-4). It is a fatal error if *F* is not known. The phototypesetter has four fonts physically mounted. Each font consists of a film strip which can be mounted on a numbered quadrant of a wheel. The default mounting sequence assumed by TROFF is R, I, B, and S on positions 1, 2, 3 and 4.

3. Page control

Top and bottom margins are *not* automatically provided; it is conventional to define two *macros* and to set *traps* for them at vertical positions 0 (top) and $-N$ (N from the bottom). See §7 and Tutorial Examples §T2. A pseudo-page transition onto the *first* page occurs either when the first *break* occurs or when the first *non-diverted* text processing occurs. Arrangements for a trap to occur at the top of the first page must be completed before this transition. In the following, references to the *current diversion* (§7.4) mean that the mechanism being described works during both ordinary and diverted output (the former considered as the top diversion level).

The useable page width on the Graphic Systems phototypesetter is about 7.54 inches, beginning about 1/27 inch from the left edge of the 8 inch wide, continuous roll paper. The physical limitations on NROFF output are output-device dependent.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.pl $\pm N$	11 in	11 in	v	Page length set to $\pm N$. The internal limitation is about 75 inches in TROFF and about 136 inches in NROFF. The current page length is available in the **.p** register.
.bp $\pm N$	N=1	-	B*,v	Begin page. The current page is ejected and a new page is begun. If $\pm N$ is given, the new page number will be $\pm N$. Also see request **ns**.
.pn $\pm N$	N=1	ignored	-	Page number. The next page (when it occurs) will have the page number $\pm N$. A **pn** must occur before the initial pseudo-page transition to effect the page number of the first page. The current page number is in the % register.
.po $\pm N$	0; 26/27 in†	previous	v	Page offset. The current *left margin* is set to $\pm N$. The TROFF initial value provides about 1 inch of paper margin including the physical typesetter margin of 1/27 inch. In TROFF the maximum (line-length) + (page-offset) is about 7.54 inches. See §6. The current page offset is available in the **.o** register.
.ne N	-	N=1 V	D,v	Need N vertical space. If the distance, D, to the next trap position (see §7.5) is less than N, a forward vertical space of size D occurs, which will spring the trap. If there are no remaining traps on the page, D is the

*The use of "'" as control character (instead of ".") suppresses the break function.

†Values separated by ":" are for NROFF and TROFF respectively.

				distance to the bottom of the page. If $D < V$, another line could still be output and spring the trap. In a diversion, D is the distance to the *diversion trap*, if any, or is very large.
.mk *R*	none	internal	D	Mark the *current* vertical place in an internal register (both associated with the current diversion level), or in register *R*, if given. See rt request.
.rt ±*N*	none	internal	D,v	Return *upward only* to a marked vertical place in the current diversion. If ±*N* (w.r.t. current place) is given, the place is ±*N* from the top of the page or diversion or, if *N* is absent, to a place marked by a previous mk. Note that the sp request (§5.3) may be used in all cases instead of rt by spacing to the absolute place stored in a explicit register; e. g. using the sequence .mk *R* ... .sp |\n*R*u.

4. Text Filling, Adjusting, and Centering

4.1. Filling and adjusting. Normally, words are collected from input text lines and assembled into a output text line until some word doesn't fit. An attempt is then made the hyphenate the word in effort to assemble a part of it into the output line. The spaces between the words on the output line are then increased to spread out the line to the current *line length* minus any current *indent*. A *word* is any string of characters delimited by the *space* character or the beginning/end of the input line. Any adjacent pair of words that must be kept together (neither split across output lines nor spread apart in the adjustment process) can be tied together by separating them with the *unpaddable space* character "\ " (backslash-space). The adjusted word spacings are uniform in TROFF and the minimum interword spacing can be controlled with the ss request (§2). In NROFF, they are normally nonuniform because of quantization to character-size spaces; however, the command line option −e causes uniform spacing with full output device resolution. Filling, adjustment, and hyphenation (§13) can all be prevented or controlled. The *text length* on the last line output is available in the .n register, and text base-line position on the page for this line is in the nl register. The text base-line high-water mark (lowest place) on the current page is in the .h register.

An input text line ending with ., ?, or ! is taken to be the end of a *sentence*, and an additional space character is automatically provided during filling. Multiple inter-word space characters found in the input are retained, except for trailing spaces; initial spaces also cause a *break*.

When filling is in effect, a \p may be imbedded or attached to a word to cause a *break* at the *end* of the word and have the resulting output line *spread out* to fill the current line length.

A text input line that happens to begin with a control character can be made to not look like a control line by prefacing it with the non-printing, zero-width filler character \&. Still another way is to specify output translation of some convenient character into the control character using tr (§10.5).

4.2. Interrupted text. The copying of a input line in *nofill* (non-fill) mode can be *interrupted* by terminating the partial line with a \c. The *next* encountered input text line will be considered to be a continuation of the same line of input text. Similarly, a word within *filled* text may be interrupted by terminating the word (and line) with \c; the next encountered text will be taken as a continuation of the interrupted word. If the intervening control lines cause a break, any partial line will be forced out along with any partial word.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.br	-	-	B	Break. The filling of the line currently being collected is stopped and the line is output without adjustment. Text lines beginning with space characters and empty text lines (blank lines) also cause a break.

.fi	fill on	-	B,E	Fill subsequent output lines. The register **.u** is 1 in fill mode and 0 in nofill mode.
.nf	fill on	-	B,E	Nofill. Subsequent output lines are *neither* filled *nor* adjusted. Input text lines are copied directly to output lines *without regard* for the current line length.
.ad *c*	adj,both	adjust	E	Line adjustment is begun. If fill mode is not on, adjustment will be deferred until fill mode is back on. If the type indicator *c* is present, the adjustment type is changed as shown in the following table.

Indicator	Adjust Type
l	adjust left margin only
r	adjust right margin only
c	center
b or **n**	adjust both margins
absent	unchanged

.na	adjust	-	E	Noadjust. Adjustment is turned off; the right margin will be ragged. The adjustment type for **ad** is not changed. Output line filling still occurs if fill mode is on.
.ce *N*	off	*N*=1	B,E	Center the next *N* input text lines within the current (line-length minus indent). If *N*=0, any residual count is cleared. A break occurs after each of the *N* input lines. If the input line is too long, it will be left adjusted.

5. Vertical Spacing

5.1. Base-line spacing. The vertical spacing *(V)* between the base-lines of successive output lines can be set using the **vs** request with a resolution of 1/144 inch = 1/2 point in TROFF, and to the output device resolution in NROFF. *V* must be large enough to accommodate the character sizes on the affected output lines. For the common type sizes (9-12 points), usual typesetting practice is to set *V* to 2 points greater than the point size; TROFF default is 10-point type on a 12-point spacing (as in this document). The current *V* is available in the **.v** register. Multiple-*V* line separation (e.g. double spacing) may be requested with **ls**.

5.2. Extra line-space. If a word contains a vertically tall construct requiring the output line containing it to have extra vertical space before and/or after it, the *extra-line-space* function **\x**'*N*' can be imbedded in or attached to that word. In this and other functions having a pair of delimiters around their parameter (here '), the delimiter choice is arbitrary, except that it can't look like the continuation of a number expression for *N*. If *N* is negative, the output line containing the word will be preceded by *N* extra vertical space; if *N* is positive, the output line containing the word will be followed by *N* extra vertical space. If successive requests for extra space apply to the same line, the maximum values are used. The most recently utilized post-line extra line-space is available in the **.a** register.

5.3. Blocks of vertical space. A block of vertical space is ordinarily requested using **sp**, which honors the *no-space* mode and which does not space *past* a trap. A contiguous block of vertical space may be reserved using **sv**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.vs *N*	1/6in;12pts	previous	E,p	Set vertical base-line spacing size *V*. Transient *extra* vertical space available with **\x**'*N*' (see above).
.ls *N*	*N*=1	previous	E	*Line* spacing set to ±*N*. *N*−1 *Vs (blank lines)* are appended to each output text line. Appended blank lines are omitted, if the text or previous appended blank line

				reached a trap position.
.sp *N*	-	*N*=1 *V*	B,v	Space vertically in *either* direction. If *N* is negative, the motion is *backward* (upward) and is limited to the distance to the top of the page. Forward (downward) motion is truncated to the distance to the nearest trap. If the no-space mode is on, no spacing occurs (see ns, and rs below).
.sv *N*	-	*N*=1 *V*	v	Save a contiguous vertical block of size *N*. If the distance to the next trap is greater than *N*, *N* vertical space is output. No-space mode has *no* effect. If this distance is less than *N*, no vertical space is immediately output, but *N* is remembered for later output (see os). Subsequent sv requests will overwrite any still remembered *N*.
.os	-	-	-	Output saved vertical space. No-space mode has *no* effect. Used to finally output a block of vertical space requested by an earlier sv request.
.ns	space	-	D	No-space mode turned on. When on, the no-space mode inhibits sp requests and bp requests *without* a next page number. The no-space mode is turned off when a line of output occurs, or with rs.
.rs	space	-	D	Restore spacing. The no-space mode is turned off.
Blank text line.		-	B	Causes a break and output of a blank line exactly like sp 1.

6. Line Length and Indenting

The maximum line length for fill mode may be set with ll. The indent may be set with in; an indent applicable to *only* the *next* output line may be set with ti. The line length includes indent space but *not* page offset space. The line-length minus the indent is the basis for centering with ce. The effect of ll, in, or ti is delayed, if a partially collected line exists, until after that line is output. In fill mode the length of text on an output line is less than or equal to the line length minus the indent. The current line length and indent are available in registers .l and .i respectively. The length of *three-part titles* produced by tl (see §14) is *independently* set by lt.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ll ±*N*	6.5 in	previous	E,m	Line length is set to ±*N*. In TROFF the maximum (line-length) + (page-offset) is about 7.54 inches.
.in ±*N*	*N*=0	previous	B,E,m	Indent is set to ±*N*. The indent is prepended to each output line.
.ti ±*N*	-	ignored	B,E,m	Temporary indent. The *next* output text line will be indented a distance ±*N* with respect to the current indent. The resulting total indent may not be negative. The current indent is not changed.

7. Macros, Strings, Diversion, and Position Traps

7.1. Macros and strings. A *macro* is a named set of arbitrary *lines* that may be invoked by name or with a *trap*. A *string* is a named string of *characters*, *not* including a newline character, that may be interpolated by name at any point. Request, macro, and string names share the *same* name list. Macro and string names may be one or two characters long and may usurp previously defined request, macro, or string names. Any of these entities may be renamed with rn or removed with rm. Macros are created by de and di, and appended to by am and da; di and da cause normal output to be stored in a macro. Strings are created by ds and appended to by as. A macro is invoked in the same way as a request; a

control line beginning *.xx* will interpolate the contents of macro *xx*. The remainder of the line may contain up to nine *arguments*. The strings *x* and *xx* are interpolated at any desired point with *x and *(*xx* respectively. String references and macro invocations may be nested.

7.2. Copy mode input interpretation. During the definition and extension of strings and macros (not by diversion) the input is read in *copy mode*. The input is copied without interpretation *except* that:

- The contents of number registers indicated by \n are interpolated.
- Strings indicated by * are interpolated.
- Arguments indicated by \$ are interpolated.
- Concealed newlines indicated by \(newline) are eliminated.
- Comments indicated by \" are eliminated.
- \t and \a are interpreted as ASCII horizontal tab and SOH respectively (§9).
- \\ is interpreted as \.
- \. is interpreted as ".".

These interpretations can be suppressed by prepending a \. For example, since \\ maps into a \, \\n will copy as \n which will be interpreted as a number register indicator when the macro or string is reread.

7.3. Arguments. When a macro is invoked by name, the remainder of the line is taken to contain up to nine arguments. The argument separator is the space character, and arguments may be surrounded by double-quotes to permit imbedded space characters. Pairs of double-quotes may be imbedded in double-quoted arguments to represent a single double-quote. If the desired arguments won't fit on a line, a concealed newline may be used to continue on the next line.

When a macro is invoked the *input level* is *pushed down* and any arguments available at the previous level become unavailable until the macro is completely read and the previous level is restored. A macro's own arguments can be interpolated at *any* point within the macro with \$*N*, which interpolates the *N*th argument ($1 \leqslant N \leqslant 9$). If an invoked argument doesn't exist, a null string results. For example, the macro *xx* may be defined by

```
.de xx      \"begin definition
Today is \\$1 the \\$2.
..          \"end definition
```

and called by

```
.xx Monday 14th
```

to produce the text

Today is Monday the 14th.

Note that the \$ was concealed in the definition with a prepended \. The number of currently available arguments is in the **.$** register.

No arguments are available at the top (non-macro) level in this implementation. Because string referencing is implemented as a input-level push down, no arguments are available from *within* a string. No arguments are available within a trap-invoked macro.

Arguments are copied in *copy mode* onto a stack where they are available for reference. The mechanism does not allow an argument to contain a direct reference to a *long* string (interpolated at copy time) and it is advisable to conceal string references (with an extra \) to delay interpolation until argument reference time.

7.4. Diversions. Processed output may be diverted into a macro for purposes such as footnote processing (see Tutorial §T5) or determining the horizontal and vertical size of some text for conditional changing of pages or columns. A single diversion trap may be set at a specified vertical position. The number registers **dn** and **dl** respectively contain the vertical and horizontal size of the most recently ended diversion. Processed text that is diverted into a macro retains the vertical size of each of its lines when reread in *nofill* mode regardless of the current *V*. Constant-spaced (**cs**) or emboldened (**bd**) text that is diverted can be reread correctly only if these modes are again or still in effect at reread time. One way

to do this is to imbed in the diversion the appropriate **cs** or **bd** requests with the *transparent* mechanism described in §10.6.

Diversions may be nested and certain parameters and registers are associated with the current diversion level (the top non-diversion level may be thought of as the 0th diversion level). These are the diversion trap and associated macro, no-space mode, the internally-saved marked place (see **mk** and **rt**), the current vertical place (**.d** register), the current high-water text base-line (**.h** register), and the current diversion name (**.z** register).

7.5. Traps. Three types of trap mechanisms are available—page traps, a diversion trap, and an input-line-count trap. Macro-invocation traps may be planted using **wh** at any page position including the top. This trap position may be changed using **ch**. Trap positions at or below the bottom of the page have no effect unless or until moved to within the page or rendered effective by an increase in page length. Two traps may be planted at the *same* position only by first planting them at different positions and then moving one of the traps; the first planted trap will conceal the second unless and until the first one is moved (see Tutorial Examples §T5). If the first one is moved back, it again conceals the second trap. The macro associated with a page trap is automatically invoked when a line of text is output whose vertical size *reaches* or *sweeps past* the trap position. Reaching the bottom of a page springs the top-of-page trap, if any, provided there is a next page. The distance to the next trap position is available in the **.t** register; if there are no traps between the current position and the bottom of the page, the distance returned is the distance to the page bottom.

A macro-invocation trap effective in the current diversion may be planted using **dt**. The **.t** register works in a diversion; if there is no subsequent trap a *large* distance is returned. For a description of input-line-count traps, see it below.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.de *xx yy*	-	*.yy*=..	-	Define or redefine the macro *xx*. The contents of the macro begin on the next input line. Input lines are copied in *copy mode* until the definition is terminated by a line beginning with *.yy*, whereupon the macro *yy* is called. In the absence of *yy*, the definition is terminated by a line beginning with "..". A macro may contain **de** requests provided the terminating macros differ or the contained definition terminator is concealed. ".." can be concealed as \\.. which will copy as \.. and be reread as "..".
.am *xx yy*	-	*.yy*=..	-	Append to macro (append version of **de**).
.ds *xx string*	-	ignored	-	Define a string *xx* containing *string*. Any initial double-quote in *string* is stripped off to permit initial blanks.
.as *xx string*	-	ignored	-	Append *string* to string *xx* (append version of **ds**).
.rm *xx*	-	ignored	-	Remove request, macro, or string. The name *xx* is removed from the name list and any related storage space is freed. Subsequent references will have no effect.
.rn *xx yy*	-	ignored	-	Rename request, macro, or string *xx* to *yy*. If *yy* exists, it is first removed.
.di *xx*	-	end	D	Divert output to macro *xx*. Normal text processing occurs during diversion except that page offsetting is not done. The diversion ends when the request **di** or **da** is encountered without an argument; extraneous requests of this type should not appear when nested diversions are being used.

.da *xx*	-	end	D	Divert, appending to *xx* (append version of **di**).
.wh *N xx*	-	-	v	Install a trap to invoke *xx* at page position *N*; a *negative N* will be interpreted with respect to the page *bottom*. Any macro previously planted at *N* is replaced by *xx*. A zero *N* refers to the *top* of a page. In the absence of *xx*, the first found trap at *N*, if any, is removed.
.ch *xx N*	-	-	v	Change the trap position for macro *xx* to be *N*. In the absence of *N*, the trap, if any, is removed.
.dt *N xx*	-	off	D,v	Install a diversion trap at position *N* in the *current* diversion to invoke macro *xx*. Another **dt** will redefine the diversion trap. If no arguments are given, the diversion trap is removed.
.it *N xx*	-	off	E	Set an input-line-count trap to invoke the macro *xx* after *N* lines of *text* input have been read (control or request lines don't count). The text may be in-line text or text interpolated by inline or trap-invoked macros.
.em *xx*	none	none	-	The macro *xx* will be invoked when all input has ended. The effect is the same as if the contents of *xx* had been at the end of the last file processed.

8. Number Registers

A variety of parameters are available to the user as predefined, named *number registers* (see Summary and Index, page 7). In addition, the user may define his own named registers. Register names are one or two characters long and *do not* conflict with request, macro, or string names. Except for certain predefined read-only registers, a number register can be read, written, automatically incremented or decremented, and interpolated into the input in a variety of formats. One common use of user-defined registers is to automatically number sections, paragraphs, lines, etc. A number register may be used any time numerical input is expected or desired and may be used in numerical *expressions* (§1.4).

Number registers are created and modified using **nr**, which specifies the name, numerical value, and the auto-increment size. Registers are also modified, if accessed with an auto-incrementing sequence. If the registers *x* and *xx* both contain *N* and have the auto-increment size *M*, the following access sequences have the effect shown:

Sequence	Effect on Register	Value Interpolated
\n*x*	none	*N*
\n(*xx*	none	*N*
\n+*x*	*x* incremented by *M*	*N+M*
\n−*x*	*x* decremented by *M*	*N−M*
\n+(*xx*	*xx* incremented by *M*	*N+M*
\n−(*xx*	*xx* decremented by *M*	*N−M*

When interpolated, a number register is converted to decimal (default), decimal with leading zeros, lower-case Roman, upper-case Roman, lower-case sequential alphabetic, or upper-case sequential alphabetic according to the format specified by **af**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.nr *R ±N M*		-	u	The number register *R* is assigned the value *±N* with respect to the previous value, if any. The increment for auto-incrementing is set to *M*.

.af *R c* arabic - - Assign format *c* to register *R*. The available formats are:

Format	Numbering Sequence
1	0,1,2,3,4,5,...
001	000,001,002,003,004,005,...
i	0,i,ii,iii,iv,v,...
I	0,I,II,III,IV,V,...
a	0,a,b,c,...,z,aa,ab,...,zz,aaa,...
A	0,A,B,C,...,Z,AA,AB,...,ZZ,AAA,...

An arabic format having *N* digits specifies a field width of *N* digits (example 2 above). The read-only registers and the *width* function (§11.2) are always arabic.

.rr *R* - ignored - Remove register *R*. If many registers are being created dynamically, it may become necessary to remove no longer used registers to recapture internal storage space for newer registers.

9. Tabs, Leaders, and Fields

9.1. Tabs and leaders. The ASCII horizontal tab character and the ASCII SOH (hereafter known as the *leader* character) can both be used to generate either horizontal motion or a string of repeated characters. The length of the generated entity is governed by internal *tab stops* specifiable with ta. The default difference is that tabs generate motion and leaders generate a string of periods; te and le offer the choice of repeated character or motion. There are three types of internal tab stops—*left* adjusting, *right* adjusting, and *centering*. In the following table: *D* is the distance from the current position on the *input* line (where a tab or leader was found) to the next tab stop; *next-string* consists of the input characters following the tab (or leader) up to the next tab (or leader) or end of line; and *W* is the width of *next-string*.

Tab type	Length of motion or repeated characters	Location of *next-string*
Left	*D*	Following *D*
Right	*D*−*W*	Right adjusted within *D*
Centered	*D*−*W*/2	Centered on right end of *D*

The length of generated motion is allowed to be negative, but that of a repeated character string cannot be. Repeated character strings contain an integer number of characters, and any residual distance is prepended as motion. Tabs or leaders found after the last tab stop are ignored, but may be used as *next-string* terminators.

Tabs and leaders are not interpreted in *copy mode*. \t and \a always generate a non-interpreted tab and leader respectively, and are equivalent to actual tabs and leaders in *copy mode*.

9.2. Fields. A *field* is contained between a *pair* of *field delimiter* characters, and consists of sub-strings separated by *padding* indicator characters. The field length is the distance on the *input* line from the position where the field begins to the next tab stop. The difference between the total length of all the sub-strings and the field length is incorporated as horizontal padding space that is divided among the indicated padding places. The incorporated padding is allowed to be negative. For example, if the field delimiter is # and the padding indicator is ^, #^xxx^*right*# specifies a right-adjusted string with the string *xxx* centered in the remaining space.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ta *Nt* ...	0.8; 0.5in	none	E,m	Set tab stops and types. *t*=**R**, right adjusting; *t*=**C**, centering; *t* absent, left adjusting. TROFF tab stops are preset every 0.5in.; NROFF every 0.8in. The stop values are separated by spaces, and a value preceded by **+** is treated as an increment to the previous stop value.
.tc *c*	none	none	E	The tab repetition character becomes *c*, or is removed specifying motion.
.lc *c*	.	none	E	The leader repetition character becomes *c*, or is removed specifying motion.
.fc *a b*	off	off	-	The field delimiter is set to *a*; the padding indicator is set to the *space* character or to *b*, if given. In the absence of arguments the field mechanism is turned off.

10. Input and Output Conventions and Character Translations

10.1. Input character translations. Ways of inputting the graphic character set were discussed in §2.1. The ASCII control characters horizontal tab (§9.1), SOH (§9.1), and backspace (§10.3) are discussed elsewhere. The newline delimits input lines. In addition, STX, ETX, ENQ, ACK, and BEL are accepted, and may be used as delimiters or translated into a graphic with **tr** (§10.5). *All* others are ignored.

The *escape* character \ introduces *escape sequences*—causes the following character to mean another character, or to indicate some function. A complete list of such sequences is given in the Summary and Index on page 6. \ should not be confused with the ASCII control character ESC of the same name. The escape character \ can be input with the sequence \\. The escape character can be changed with **ec**, and all that has been said about the default \ becomes true for the new escape character. \e can be used to print whatever the current escape character is. If necessary or convenient, the escape mechanism may be turned off with **eo**, and restored with **ec**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ec *c*	\	\	-	Set escape character to \, or to *c*, if given.
.eo	on	-	-	Turn escape mechanism off.

10.2. Ligatures. Five ligatures are available in the current TROFF character set — **fi**, **fl**, **ff**, **ffi**, and **ffl**. They may be input (even in NROFF) by **\(fi**, **\(fl**, **\(ff**, **\(Fi**, and **\(Fl** respectively. The ligature mode is normally on in TROFF, and *automatically* invokes ligatures during input.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.lg *N*	off; on	on	-	Ligature mode is turned on if *N* is absent or non-zero, and turned off if *N*=0. If *N*=2, only the two-character ligatures are automatically invoked. Ligature mode is inhibited for request, macro, string, register, or file names, and in *copy mode*. No effect in NROFF.

10.3. Backspacing, underlining, overstriking, etc.. Unless in *copy mode*, the ASCII backspace character is replaced by a backward horizontal motion having the width of the space character. Underlining as a form of line-drawing is discussed in §12.4. A generalized overstriking function is described in §12.1.

NROFF automatically underlines characters in the *underline* font, specifiable with **uf**, normally that on font position 2 (normally Times Italic, see §2.2). In addition to **ft** and **\f***F*, the underline font may be selected by **ul** and **cu**. Underlining is restricted to an output-device-dependent subset of *reasonable* characters.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ul *N*	off	*N*=1	E	Underline in NROFF (italicize in TROFF) the next *N* input text lines. Actually, switch to *underline* font, saving the current font for later restoration; *other* font changes within the span of a **ul** will take effect, but the restoration will undo the last change. Output generated by **tl** (§14) *is* affected by the font change, but does *not* decrement *N*. If *N*>1, there is the risk that a trap interpolated macro may provide text lines within the span; environment switching can prevent this.
.cu *N*	off	*N*=1	E	A variant of **ul** that causes *every* character to be underlined in NROFF. Identical to **ul** in TROFF.
.uf *F*	Italic	Italic	-	Underline font set to *F*. In NROFF, *F* may *not* be on position 1 (initially Times Roman).

10.4. Control characters. Both the control character . and the *no-break* control character ' may be changed, if desired. Such a change must be compatible with the design of any macros used in the span of the change, and particularly of any trap-invoked macros.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.cc *c*	.	.	E	The basic control character is set to *c*, or reset to ".".
.c2 *c*	'	'	E	The *nobreak* control character is set to *c*, or reset to "'".

10.5. Output translation. One character can be made a stand-in for another character using **tr**. All text processing (e. g. character comparisons) takes place with the input (stand-in) character which appears to have the width of the final character. The graphic translation occurs at the moment of output (including diversion).

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.tr *abcd....*	none	-	O	Translate *a* into *b*, *c* into *d*, etc. If an odd number of characters is given, the last one will be mapped into the space character. To be consistent, a particular translation must stay in effect from *input* to *output* time.

10.6. Transparent throughput. An input line beginning with a \! is read in *copy mode* and *transparently* output (without the initial \!); the text processor is otherwise unaware of the line's presence. This mechanism may be used to pass control information to a post-processor or to imbed control lines in a macro created by a diversion.

10.7. Comments and concealed newlines. An uncomfortably long input line that must stay one line (e. g. a string definition, or nofilled text) can be split into many physical lines by ending all but the last one with the escape \. The sequence \(newline) is *always* ignored—except in a comment. Comments may be imbedded at the *end* of any line by prefacing them with \". The newline at the end of a comment cannot be concealed. A line beginning with \" will appear as a blank line and behave like **.sp 1**; a comment can be on a line by itself by beginning the line with .\".

11. Local Horizontal and Vertical Motions, and the Width Function

11.1. Local Motions. The functions \v'*N*' and \h'*N*' can be used for *local* vertical and horizontal motion respectively. The distance *N* may be negative; the *positive* directions are *rightward* and *downward*. A *local* motion is one contained *within* a line. To avoid unexpected vertical dislocations, it is necessary that the *net* vertical local motion within a word in filled text and otherwise within a line balance to zero. The above and certain other escape sequences providing local motion are summarized in the following table.

Vertical Local Motion	Effect in TROFF	Effect in NROFF	Horizontal Local Motion	Effect in TROFF	Effect in NROFF
\v'N'	Move distance N		\h'N'	Move distance N	
			\(space)	Unpaddable space-size space	
\u	½ em up	½ line up	\0	Digit-size space	
\d	½ em down	½ line down			
\r	1 em up	1 line up	|	1/6 em space	ignored
			\^	1/12 em space	ignored

As an example, E^2 could be generated by the sequence **E\s-2\v'-0.4m'2\v'0.4m'\s+2**; it should be noted in this example that the 0.4 em vertical motions are at the smaller size.

11.2. Width Function. The *width* function **\w'***string***'** generates the numerical width of *string* (in basic units). Size and font changes may be safely imbedded in *string*, and will not affect the current environment. For example, **.ti -\w'1. 'u** could be used to temporarily indent leftward a distance equal to the size of the string "1. ".

The width function also sets three number registers. The registers **st** and **sb** are set respectively to the highest and lowest extent of *string* relative to the baseline; then, for example, the total *height* of the string is **\n(stu-\n(sbu**. In TROFF the number register **ct** is set to a value between 0 and 3: 0 means that all of the characters in *string* were short lower case characters without descenders (like **e**); 1 means that at least one character has a descender (like **y**); 2 means that at least one character is tall (like **H**); and 3 means that both tall characters and characters with descenders are present.

11.3. Mark horizontal place. The escape sequence **\k***x* will cause the *current* horizontal position in the *input line* to be stored in register *x*. As an example, the construction **\k***x word***\h'|\n***x***+2u'***word*** will embolden *word* by backing up to almost its beginning and overprinting it, resulting in ***word***.

12. Overstrike, Bracket, Line-drawing, and Zero-width Functions

12.1. Overstriking. Automatically centered overstriking of up to nine characters is provided by the *overstrike* function **\o'***string***'**. The characters in *string* overprinted with centers aligned; the total width is that of the widest character. *string* should *not* contain local vertical motion. As examples, **\o'e\'** produces é, and **\o'\(mo\(sl'** produces ∉.

12.2. Zero-width characters. The function **\z***c* will output *c* without spacing over it, and can be used to produce left-aligned overstruck combinations. As examples, **\z\(ci\(pl** will produce ⊕, and **\(br\z\(rn\(ul\(br** will produce the smallest possible constructed box □.

12.3. Large Brackets. The Special Mathematical Font contains a number of bracket construction pieces (⌈ ⌊ ⌉ ⌋ { } ⎪ ⎢ ⎥ ⎡ ⎤) that can be combined into various bracket styles. The function **\b'***string***'** may be used to pile up vertically the characters in *string* (the first character on top and the last at the bottom); the characters are vertically separated by 1 em and the total pile is centered 1/2 em above the current baseline (½ line in NROFF). For example, **\b'\(lc\(lf'E\|\b'\(rc\(rf'\x'-0.5m'\x'0.5m'** produces ⌈E⌉.

12.4. Line drawing. The function **\l'***Nc***'** will draw a string of repeated *c*'s towards the right for a distance *N*. (**\l** is **\(**lower case L). If *c* looks like a continuation of an expression for *N*, it may insulated from *N* with a **\&**. If *c* is not specified, the _ (baseline rule) is used (underline character in NROFF). If *N* is negative, a backward horizontal motion of size *N* is made *before* drawing the string. Any space resulting from *N*/(size of *c*) having a remainder is put at the beginning (left end) of the string. In the case of characters that are designed to be connected such as baseline-rule _, underrule ‗, and rooten ‾, the remainder space is covered by over-lapping. If *N* is *less* than the width of *c*, a single *c* is centered on a distance *N*. As an example, a macro to underscore a string can be written

```
.de us
\\$1\l'|0\(ul'
..
```

or one to draw a box around a string

```
.de bx
\(br\|\\$1\|\(br\l'|0\(rn\l'|0\(ul'
..
```

such that

```
.ul "underlined words"
```

and

```
.bx "words in a box"
```

yield underlined words and words in a box.

The function \L'*Nc*' will draw a vertical line consisting of the (optional) character *c* stacked vertically apart 1 em (1 line in NROFF), with the first two characters overlapped, if necessary, to form a continuous line. The default character is the *box rule* | (\(br); the other suitable character is the *bold vertical* | (\(bv). The line is begun without any initial motion relative to the current base line. A positive *N* specifies a line drawn downward and a negative *N* specifies a line drawn upward. After the line is drawn *no* compensating motions are made; the instantaneous baseline is at the *end* of the line.

The horizontal and vertical line drawing functions may be used in combination to produce large boxes. The zero-width *box-rule* and the ½-em wide *underrule* were *designed* to form corners when using 1-em vertical spacings. For example the macro

```
.de eb
.sp -1          \"compensate for next automatic base-line spacing
.nf             \"avoid possibly overflowing word buffer
\h'-.5n\L'|\\nau-1\l'\n(.lu+1n\(ul\L'-|\\nau+1\l'|0u-.5n\(ul'    \"draw box
.fi
..
```

will draw a box around some text whose beginning vertical place was saved in number register *a* (e. g. using .mk a) as done for this paragraph.

13. Hyphenation.

The automatic hyphenation may be switched off and on. When switched on with **hy**, several variants may be set. A *hyphenation indicator* character may be imbedded in a word to specify desired hyphenation points, or may be prepended to suppress hyphenation. In addition, the user may specify a small exception word list.

Only words that consist of a central alphabetic string surrounded by (usually null) non-alphabetic strings are considered candidates for automatic hyphenation. Words that were input containing hyphens (minus), em-dashes (\(em), or hyphenation indicator characters—such as mother-in-law—are *always* subject to splitting after those characters, whether or not automatic hyphenation is on or off.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.nh	hyphenate	-	E	Automatic hyphenation is turned off.
.hy*N*	on, *N*=1	on, *N*=1	E	Automatic hyphenation is turned on for $N \geq 1$, or off for *N*=0. If *N*=2, *last* lines (ones that will cause a trap) are not hyphenated. For *N*=4 and 8, the last and first two characters respectively of a word are not split off. These values are additive; i. e. *N*=14 will invoke all three restrictions.
.hc *c*	\%	\%	E	Hyphenation indicator character is set to *c* or to the default \%. The indicator does not appear in the output.
.hw *word1* ...		ignored	-	Specify hyphenation points in words with imbedded minus signs. Versions of a word with terminal *s* are

implied; i. e. *dig−it* implies *dig−its*. This list is examined initially *and* after each suffix stripping. The space available is small—about 128 characters.

14. Three Part Titles.

The titling function **tl** provides for automatic placement of three fields at the left, center, and right of a line with a title-length specifiable with **lt**. **tl** may be used anywhere, and is independent of the normal text collecting process. A common use is in header and footer macros.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.tl *'left'center'right'*		-	-	The strings *left*, *center*, and *right* are respectively left-adjusted, centered, and right-adjusted in the current title-length. Any of the strings may be empty, and overlapping is permitted. If the page-number character (initially %) is found within any of the fields it is replaced by the current page number having the format assigned to register %. Any character may be used as the string delimiter.
.pc *c*	%	off	-	The page number character is set to *c*, or removed. The page-number register remains %.
.lt ±*N*	6.5 in	previous	E,m	Length of title set to ±*N*. The line-length and the title-length are *independent*. Indents do not apply to titles; page-offsets do.

15. Output Line Numbering.

Automatic sequence numbering of output lines may be requested with **nm**. When in effect, a three-digit, arabic number plus a digit-space is prepended to output text lines. The text lines are thus offset by four digit-spaces, and otherwise retain their line length; a reduction in line length may be desired to keep the right margin aligned with an earlier margin. Blank lines, other vertical spaces, and lines generated by **tl** are *not* numbered. Numbering can be temporarily suspended with **nn**, or with an **.nm** followed by a later **.nm +0**. In addition, a line number indent *I*, and the number-text separation *S* may be specified in digit-spaces. Further, it can be specified that only those line numbers that are multiples of some number *M* are to be printed (the others will appear as blank number fields).

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.nm ±*N M S I*		off	E	Line number mode. If ±*N* is given, line numbering is turned on, and the next output line numbered is numbered ±*N*. Default values are *M*=1, *S*=1, and *I*=0 Parameters corresponding to missing arguments are unaffected; a non-numeric argument is considered missing. In the absence of all arguments, numbering is turned off; the next line number is preserved for possible further use in number register **ln**.
.nn *N*	-	*N*=1	E	The next *N* text output lines are not numbered.

As an example, the paragraph portions of this section are numbered with *M*=3: **.nm 1 3** was placed at the beginning; **.nm** was placed at the end of the first paragraph; and **.nm +0** was placed in front of this paragraph; and **.nm** finally placed at the end. Line lengths were also changed (by **\w'0000'u**) to keep the right side aligned. Another example is **.nm +5 5 x 3** which turns on numbering with the line number of the next line to be 5 greater than the last numbered line, with *M*=5, with spacing *S* untouched, and with the indent *I* set to 3.

16. Conditional Acceptance of Input

In the following, *c* is a one-character, built-in *condition* name, ! signifies *not*, *N* is a numerical expression, *string1* and *string2* are strings delimited by any non-blank, non-numeric character *not* in the strings, and *anything* represents what is conditionally accepted.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.if *c anything*		-	-	If condition *c* true, accept *anything* as input; in multi-line case use \{*anything*\}.
.if !*c anything*		-	-	If condition *c* false, accept *anything*.
.if *N anything*		-	u	If expression $N > 0$, accept *anything*.
.if !*N anything*		-	u	If expression $N \leq 0$, accept *anything*.
.if *'string1'string2' anything*			-	If *string1* identical to *string2*, accept *anything*.
.if !*'string1'string2' anything*			-	If *string1* not identical to *string2*, accept *anything*.
.ie *c anything*		-	u	If portion of if-else; all above forms (like **if**).
.el *anything*		-	-	Else portion of if-else.

The built-in condition names are:

Condition Name	True If
o	Current page number is odd
e	Current page number is even
t	Formatter is TROFF
n	Formatter is NROFF

If the condition *c* is *true*, or if the number *N* is greater than zero, or if the strings compare identically (including motions and character size and font), *anything* is accepted as input. If a ! precedes the condition, number, or string comparison, the sense of the acceptance is reversed.

Any spaces between the condition and the beginning of *anything* are skipped over. The *anything* can be either a single input line (text, macro, or whatever) or a number of input lines. In the multi-line case, the first line must begin with a left delimiter \{ and the last line must end with a right delimiter \}.

The request **ie** (if-else) is identical to **if** except that the acceptance state is remembered. A subsequent and matching **el** (else) request then uses the reverse sense of that state. **ie** - **el** pairs may be nested.

Some examples are:

```
.if e .tl 'Even Page %'''
```

which outputs a title if the page number is even; and

```
.ie \n%>1 \{\
'sp 0.5i
.tl 'Page %'''
'sp |1.2i \}
.el .sp |2.5i
```

which treats page 1 differently from other pages.

17. Environment Switching.

A number of the parameters that control the text processing are gathered together into an *environment*, which can be switched by the user. The environment parameters are those associated with requests noting E in their *Notes* column; in addition, partially collected lines and words are in the environment. Everything else is global; examples are page-oriented parameters, diversion-oriented parameters,

number registers, and macro and string definitions. All environments are initialized with default parameter values.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ev *N*	*N*=0	previous	-	Environment switched to environment $0 \leq N \leq 2$. Switching is done in push-down fashion so that restoring a previous environment *must* be done with .ev rather than specific reference.

18. Insertions from the Standard Input

The input can be temporarily switched to the system *standard input* with **rd**, which will switch back when *two* newlines in a row are found (the *extra* blank line is not used). This mechanism is intended for insertions in form-letter-like documentation. On UNIX, the *standard input* can be the user's keyboard, a *pipe*, or a *file*.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.rd *prompt*	-	*prompt*=BEL	-	Read insertion from the standard input until two newlines in a row are found. If the standard input is the user's keyboard, *prompt* (or a BEL) is written onto the user's terminal. **rd** behaves like a macro, and arguments may be placed after *prompt*.
.ex	-	-	-	Exit from NROFF/TROFF. Text processing is terminated exactly as if all input had ended.

If insertions are to be taken from the terminal keyboard *while* output is being printed on the terminal, the command line option **−q** will turn off the echoing of keyboard input and prompt only with BEL. The regular input and insertion input *cannot* simultaneously come from the standard input.

As an example, multiple copies of a form letter may be prepared by entering the insertions for all the copies in one file to be used as the standard input, and causing the file containing the letter to reinvoke itself using **nx** (§19); the process would ultimately be ended by an **ex** in the insertion file.

19. Input/Output File Switching

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.so *filename*		-	-	Switch source file. The top input (file reading) level is switched to *filename*. The effect of an **so** encountered in a macro is not felt until the input level returns to the file level. When the new file ends, input is again taken from the original file. **so**'s may be nested.
.nx *filename*		end-of-file	-	Next file is *filename*. The current file is considered ended, and the input is immediately switched to *filename*.
.pi *program*		-	-	Pipe output to *program* (NROFF only). This request must occur *before* any printing occurs. No arguments are transmitted to *program*.

20. Miscellaneous

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.mc *c N*	-	off	E,m	Specifies that a *margin* character *c* appear a distance *N* to the right of the right margin after each non-empty text line (except those produced by **tl**). If the output line is too-long (as can happen in nofill mode) the character will

				be appended to the line. If *N* is not given, the previou *N* is used; the initial *N* is 0.2 inches in NROFF and 1 en in TROFF. The margin character used with this para graph was a 12-point box-rule.
.tm *string*	-	newline	-	After skipping initial blanks, *string* (rest of the line) i read in *copy mode* and written on the user's terminal.
.ig *yy*	-	.*yy*=..	-	Ignore input lines. ig behaves exactly like de (§7) excep that the input is discarded. The input is read in *copy mode*, and any auto-incremented registers will be affected.
.pm *t*	-	all	-	Print macros. The names and sizes of all of the defined macros and strings are printed on the user's terminal; if *t* is given, only the total of the sizes is printed. The sizes is given in *blocks* of 128 characters.
.fl	-	-	B	Flush output buffer. Used in interactive debugging to force output.

21. Output and Error Messages.

The output from tm, pm, and the prompt from rd, as well as various *error* messages are written onto UNIX's *standard message* output. The latter is different from the *standard output*, where NROFF formatted output goes. By default, both are written onto the user's terminal, but they can be independently redirected.

Various *error* conditions may occur during the operation of NROFF and TROFF. Certain less serious errors having only local impact do not cause processing to terminate. Two examples are *word overflow*, caused by a word that is too large to fit into the word buffer (in fill mode), and *line overflow*, caused by an output line that grew too large to fit in the line buffer; in both cases, a message is printed, the offending excess is discarded, and the affected word or line is marked at the point of truncation with a • in NROFF and a ▬ in TROFF. The philosophy is to continue processing, if possible, on the grounds that output useful for debugging may be produced. If a serious error occurs, processing terminates, and an appropriate message is printed. Examples are the inability to create, read, or write files, and the exceeding of certain internal limits that make future output unlikely to be useful.

TUTORIAL EXAMPLES

T1. Introduction

Although NROFF and TROFF have by design a syntax reminiscent of earlier text processors* with the intent of easing their use, it is almost always necessary to prepare at least a small set of macro definitions to describe most documents. Such common formatting needs as page margins and footnotes are deliberately not built into NROFF and TROFF. Instead, the macro and string definition, number register, diversion, environment switching, page-position trap, and conditional input mechanisms provide the basis for user-defined implementations.

The examples to be discussed are intended to be useful and somewhat realistic, but won't necessarily cover all relevant contingencies. Explicit numerical parameters are used in the examples to make them easier to read and to illustrate typical values. In many cases, number registers would really be used to reduce the number of places where numerical information is kept, and to concentrate conditional parameter initialization like that which depends on whether TROFF or NROFF is being used.

T2. Page Margins

As discussed in §3, *header* and *footer* macros are usually defined to describe the top and bottom page margin areas respectively. A trap is planted at page position 0 for the header, and at $-N$ (N from the page bottom) for the footer. The simplest such definitions might be

```
.de hd          \"define header
'sp 1i
..              \"end definition
.de fo          \"define footer
'bp
..              \"end definition
.wh 0 hd
.wh -1i fo
```

which provide blank 1 inch top and bottom margins. The header will occur on the *first* page, only if the definition and trap exist prior to the initial pseudo-page transition (§3). In fill mode, the output line that springs the footer trap was typically forced out because some part or whole word didn't fit on it. If anything in the footer and header that follows causes a *break*, that word or part word will be forced out. In this and other examples, requests like **bp** and **sp** that normally cause breaks are invoked using the *no-break* control character ' to avoid this. When the header/footer design contains material requiring independent text processing, the environment may be switched, avoiding most interaction with the running text.

A more realistic example would be

```
.de hd          \"header
.if t .tl ''\(rn''\(rn'' \"troff cut mark
.if \\n%>1 \{\
'sp |0.5i-1     \"tl base at 0.5i
.tl ''- % -''   \"centered page number
.ps             \"restore size
.ft             \"restore font
.vs \}          \"restore vs
'sp |1.0i       \"space to 1.0i
.ns             \"turn on no-space mode
..
.de fo          \"footer
.ps 10          \"set footer/header size
.ft R           \"set font
.vs 12p         \"set base-line spacing
.if \\n%=1 \{\
'sp |\\n(.pu-0.5i-1  \"tl base 0.5i up
.tl ''- % -''\}  \"first page number
'bp
..
.wh 0 hd
.wh -1i fo
```

which sets the size, font, and base-line spacing for the header/footer material, and ultimately restores them. The material in this case is a page number at the bottom of the first page and at the top of the remaining pages. If TROFF is used, a *cut mark* is drawn in the form of *root-en*'s at each margin. The **sp**'s refer to absolute positions to avoid dependence on the base-line spacing. Another reason for this in the footer is that the footer is invoked by printing a line whose vertical spacing swept past the trap position by possibly as

*For example: P. A. Crisman, Ed., *The Compatible Time-Sharing System*, MIT Press, 1965, Section AH9.01 (Description of RUNOFF program on MIT's CTSS system).

much as the base-line spacing. The *no-space* mode is turned on at the end of **hd** to render ineffective accidental occurrences of **sp** at the top of the running text.

The above method of restoring size, font, etc. presupposes that such requests (that set *previous* value) are *not* used in the running text. A better scheme is save and restore both the current *and* previous values as shown for size in the following:

```
.de fo
.nr s1 \\n(.s     \"current size
.ps
.nr s2 \\n(.s     \"previous size
. —               \"rest of footer
..
.de hd
. —               \"header stuff
.ps \\n(s2        \"restore previous size
.ps \\n(s1        \"restore current size
..
```

Page numbers may be printed in the bottom margin by a separate macro triggered during the footer's page ejection:

```
.de bn            \"bottom number
.tl "— % —"       \"centered page number
..
.wh -0.5i-1v bn \"tl base 0.5i up
```

T3. Paragraphs and Headings

The housekeeping associated with starting a new paragraph should be collected in a paragraph macro that, for example, does the desired preparagraph spacing, forces the correct font, size, base-line spacing, and indent, checks that enough space remains for *more than one* line, and requests a temporary indent.

```
.de pg            \"paragraph
.br               \"break
.ft R             \"force font,
.ps 10            \"size,
.vs 12p           \"spacing,
.in 0             \"and indent
.sp 0.4           \"prespace
.ne 1+\\n(.Vu     \"want more than 1 line
.ti 0.2i          \"temp indent
..
```

The first break in **pg** will force out any previous partial lines, and must occur before the **vs**. The forcing of font, etc. is partly a defense against prior error and partly to permit things like section heading macros to set parameters only once. The prespacing parameter is suitable for TROFF; a larger space, at least as big as the output device vertical resolution, would be more suitable in NROFF. The choice of remaining space to test for in the **ne** is the smallest amount greater than one line (the .V is the available vertical resolution).

A macro to automatically number section headings might look like:

```
.de sc            \"section
. —               \"force font, etc.
.sp 0.4           \"prespace
.ne 2.4+\\n(.Vu \"want 2.4+ lines
.fi
\\n+S.
..
.nr S 0 1         \"init S
```

The usage is .sc, followed by the section heading text, followed by **.pg**. The **ne** test value includes one line of heading, 0.4 line in the following **pg**, and one line of the paragraph text. A word consisting of the next section number and a period is produced to begin the heading line. The format of the number may be set by **af** (§8).

Another common form is the labeled, indented paragraph, where the label protrudes left into the indent space.

```
.de lp            \"labeled paragraph
.pg
.in 0.5i          \"paragraph indent
.ta 0.2i 0.5i     \"label, paragraph
.ti 0
\t\\$1\t\c        \"flow into paragraph
..
```

The intended usage is ".lp *label*"; *label* will begin at 0.2 inch, and cannot exceed a length of 0.3 inch without intruding into the paragraph. The label could be right adjusted against 0.4 inch by setting the tabs instead with **.ta 0.4iR 0.5i**. The last line of **lp** ends with \c so that it will become a part of the first line of the text that follows.

T4. Multiple Column Output

The production of multiple column pages requires the footer macro to decide whether it was invoked by other than the last column, so that it will begin a new column rather than produce the bottom margin. The header can initialize a column register that the footer will increment and test. The following is arranged for two columns, but is easily modified for more.

```
.de hd          \"header
. ---
.nr cl 0 1      \"init column count
.mk             \"mark top of text
..
.de fo          \"footer
.ie \\n+(cl<2 \{\
.po +3.4i       \"next column: 3.1+0.3
.rt             \"back to mark
.ns \}          \"no-space mode
.el \{\
.po \\nMu       \"restore left margin
. ---
'bp \}
..
.ll 3.1i        \"column width
.nr M \\n(.o    \"save left margin
```

Typically a portion of the top of the first page contains full width text; the request for the narrower line length, as well as another **.mk** would be made where the two column output was to begin.

T5. Footnote Processing

The footnote mechanism to be described is used by imbedding the footnotes in the input text at the point of reference, demarcated by an initial **.fn** and a terminal **.ef**:

```
.fn
Footnote text and control lines...
.ef
```

In the following, footnotes are processed in a separate environment and diverted for later printing in the space immediately prior to the bottom margin. There is provision for the case where the last collected footnote doesn't completely fit in the available space.

```
.de hd          \"header
. ---
.nr x 0 1       \"init footnote count
.nr y 0-\\nb    \"current footer place
.ch fo -\\nbu   \"reset footer trap
.if \\n(dn .fz  \"leftover footnote
..
.de fo          \"footer
.nr dn 0        \"zero last diversion size
.if \\nx \{\
.ev 1           \"expand footnotes in ev1
.nf             \"retain vertical size
.FN             \"footnotes
.rm FN          \"delete it
.if "\\n(.z"fy" .di \"end overflow diversion
.nr x 0         \"disable fx
.ev \}          \"pop environment
. ---
'bp
..
.de fx          \"process footnote overflow
.if \\nx .di fy \"divert overflow
..
.de fn          \"start footnote
.da FN          \"divert (append) footnote
.ev 1           \"in environment 1
.if \\n+x=1 .fs \"if first, include separator
.fi             \"fill mode
..
.de ef          \"end footnote
.br             \"finish output
.nr z \\n(.v    \"save spacing
.ev             \"pop ev
.di             \"end diversion
.nr y -\\n(dn   \"new footer position,
.if \\nx=1 .nr y -(\\n(.v-\\nz) \
                \"uncertainty correction
.ch fo \\nyu    \"y is negative
.if (\\n(nl+1v)>(\\n(.p+\\ny) \
.ch fo \\n(nlu+1v \"it didn't fit
..
.de fs          \"separator
\l' 1i'         \"1 inch rule
.br
..
.de fz          \"get leftover footnote
.fn
.nf             \"retain vertical size
.fy             \"where fx put it
.ef
..
.nr b 1.0i      \"bottom margin size
.wh 0 hd        \"header trap
.wh 12i fo      \"footer trap, temp position
.wh -\\nbu fx   \"fx at footer position
.ch fo -\\nbu   \"conceal fx with fo
```

The header **hd** initializes a footnote count register **x**, and sets both the current footer trap position register **y** and the footer trap itself to a nominal position specified in register **b**. In addition, if the register **dn** indicates a leftover footnote, **fz** is invoked to reprocess it. The footnote start macro **fn** begins a diversion (append) in environment 1, and increments the count **x**; if the count is one, the footnote separator **fs** is interpolated. The separator is kept in a separate macro to permit user redefinition. The footnote end macro **ef** restores the previous environment and ends the diversion after saving the spacing size in register **z**. **y** is then decremented by the size of the

footnote, available in **dn**; then on the first footnote, **y** is further decremented by the difference in vertical base-line spacings of the two environments, to prevent the late triggering the footer trap from causing the last line of the combined footnotes to overflow. The footer trap is then set to the lower (on the page) of **y** or the current page position (**nl**) plus one line, to allow for printing the reference line. If indicated by **x**, the footer **fo** rereads the footnotes from **FN** in nofill mode in environment 1, and deletes **FN**. If the footnotes were too large to fit, the macro **fx** will be trap-invoked to redivert the overflow into **fy**, and the register **dn** will later indicate to the header whether **fy** is empty. Both **fo** and **fx** are planted in the nominal footer trap position in an order that causes **fx** to be concealed unless the **fo** trap is moved. The footer then terminates the overflow diversion, if necessary, and zeros **x** to disable **fx**, because the uncertainty correction together with a not-too-late triggering of the footer can result in the footnote rereading finishing before reaching the **fx** trap.

A good exercise for the student is to combine the multiple-column and footnote mechanisms.

T6. The Last Page

After the last input file has ended, NROFF and TROFF invoke the *end macro* (§7), if any, and when it finishes, eject the remainder of the page. During the eject, any traps encountered are processed normally. At the *end* of this last page, processing terminates *unless* a partial line, word, or partial word remains. If it is desired that another page be started, the end-macro

```
.de en          \"end-macro
\c
'bp
..
.em en
```

will deposit a null partial word, and effect another last page.

Table I

Font Style Examples

The following fonts are printed in 12-point, with a vertical spacing of 14-point, and with non-alphanumeric characters separated by ¼ em space. The Special Mathematical Font was specially prepared for Bell Laboratories by Graphic Systems, Inc. of Hudson, New Hampshire. The Times Roman, Italic, and Bold are among the many standard fonts available from that company.

Times Roman

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |
● □ — - _ ¼ ½ ¾ fi fl ff ffi ffl ° † ’ ¢ ® ©

Times Italic

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
*! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |*
● □ — - _ ¼ ½ ¾ fi fl ff ffi ffl ° † ’ ¢ ® ©

Times Bold

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |
● □ — - _ ¼ ½ ¾ fi fl ff ffi ffl ° † ’ ¢ ® ©

Special Mathematical Font

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

Table II

Input Naming Conventions for ´, `,and — and for Non-ASCII Special Characters

Non-ASCII characters and *minus* on the standard fonts.

Char	*Input Name*	*Character Name*
'	´	close quote
`	`	open quote
—	\(em	3/4 Em dash
-	—	hyphen or
-	\(hy	hyphen
−	\−	current font minus
•	\(bu	bullet
□	\(sq	square
_	\(ru	rule
¼	\(14	1/4
½	\(12	1/2
¾	\(34	3/4

Char	*Input Name*	*Character Name*
fi	\(fi	fi
fl	\(fl	fl
ff	\(ff	ff
ffi	\(Fi	ffi
ffl	\(Fl	ffl
°	\(de	degree
†	\(dg	dagger
'	\(fm	foot mark
¢	\(ct	cent sign
®	\(rg	registered
©	\(co	copyright

Non-ASCII characters and ´, `, _, +, −, =, and • on the special font.

The ASCII characters @, #, ", ´, `, <, >, \, {, }, ~, ^, and _ exist *only* on the special font and are printed as a 1-em space if that font is not mounted. The following characters exist only on the special font except for the upper case Greek letter names followed by † which are mapped into upper case English letters in whatever font is mounted on font position one (default Times Roman). The special math plus, minus, and equals are provided to insulate the appearance of equations from the choice of standard fonts.

Char	*Input Name*	*Character Name*
+	\(pl	math plus
−	\(mi	math minus
=	\(eq	math equals
•	\(**	math star
§	\(sc	section
´	\(aa	acute accent
`	\(ga	grave accent
_	\(ul	underrule
/	\(sl	slash (matching backslash)
α	\(*a	alpha
β	\(*b	beta
γ	\(*g	gamma
δ	\(*d	delta
ε	\(*e	epsilon
ζ	\(*z	zeta
η	\(*y	eta
θ	\(*h	theta
ι	\(*i	iota

Char	*Input Name*	*Character Name*
κ	\(*k	kappa
λ	\(*l	lambda
μ	\(*m	mu
ν	\(*n	nu
ξ	\(*c	xi
ο	\(*o	omicron
π	\(*p	pi
ρ	\(*r	rho
σ	\(*s	sigma
ς	\(ts	terminal sigma
τ	\(*t	tau
υ	\(*u	upsilon
φ	\(*f	phi
χ	\(*x	chi
ψ	\(*q	psi
ω	\(*w	omega
A	\(*A	Alpha†
B	\(*B	Beta†

Char	Input Name	Character Name
Γ	\(*G	Gamma
Δ	\(*D	Delta
E	\(*E	Epsilon†
Z	\(*Z	Zeta†
H	\(*Y	Eta†
Θ	\(*H	Theta
I	\(*I	Iota†
K	\(*K	Kappa†
Λ	\(*L	Lambda
M	\(*M	Mu†
N	\(*N	Nu†
Ξ	\(*C	Xi
O	\(*O	Omicron†
Π	\(*P	Pi
P	\(*R	Rho†
Σ	\(*S	Sigma
T	\(*T	Tau†
Y	\(*U	Upsilon
Φ	\(*F	Phi
X	\(*X	Chi†
Ψ	\(*Q	Psi
Ω	\(*W	Omega
√	\(sr	square root
	\(rn	root en extender
≥	\(>=	>=
≤	\(<=	<=
≡	\(==	identically equal
≃	\(~=	approx =
~	\(ap	approximates
≠	\(!=	not equal
→	\(->	right arrow
←	\(<-	left arrow
↑	\(ua	up arrow
↓	\(da	down arrow
×	\(mu	multiply
÷	\(di	divide
±	\(+-	plus-minus
∪	\(cu	cup (union)
∩	\(ca	cap (intersection)
⊂	\(sb	subset of
⊃	\(sp	superset of
⊆	\(ib	improper subset
⊇	\(ip	improper superset
∞	\(if	infinity
∂	\(pd	partial derivative
∇	\(gr	gradient
¬	\(no	not
∫	\(is	integral sign
∝	\(pt	proportional to
∅	\(es	empty set
∈	\(mo	member of

Char	Input Name	Character Name
|	\(br	box vertical rule
‡	\(dd	double dagger
☛	\(rh	right hand
☚	\(lh	left hand
Ⓑ	\(bs	Bell System logo
|	\(or	or
○	\(ci	circle
⎧	\(lt	left top of big curly bracket
⎩	\(lb	left bottom
⎫	\(rt	right top
⎭	\(rb	right bot
⎨	\(lk	left center of big curly bracket
⎬	\(rk	right center of big curly bracket
▯	\(bv	bold vertical
⌊	\(lf	left floor (left bottom of big square bracket)
⌋	\(rf	right floor (right bottom)
⌈	\(lc	left ceiling (left top)
⌉	\(rc	right ceiling (right top)

May 15, 1977

Summary of Changes to N/TROFF Since October 1976 Manual

Options

-h — (Nroff only) Output tabs used during horizontal spacing to speed output as well as reduce output byte count. Device tab settings assumed to be every 8 nominal character widths. The default settings of input (logical) tabs is also initialized to every 8 nominal character widths.

-z — Efficiently suppresses formatted output. Only message output will occur (from "tm"s and diagnostics).

Old Requests

.ad c — The adjustment type indicator "c" may now also be a number previously obtained from the ".j" register (see below).

.so name — The contents of file "name" will be interpolated at the point the "so" is encountered. Previously, the interpolation was done upon return to the file-reading input level.

New Request

.ab text — Prints "text" on the message output and terminates without further processing. If "text" is missing, "User Abort." is printed. Does not cause a break. The output buffer is flushed.

.fz F N — forces font "F" to be in size N. N may have the form N, +N, or -N. For example,

```
.fz 3 -2
```

will cause an implicit \s-2 every time font 3 is entered, and a corresponding \s+2 when it is left. Special font characters occurring during the reign of font F will have the same size modification. If special characters are to be treated differently,

```
.fz S F N
```

may be used to specify the size treatment of special characters during font F. For example,

```
.fz 3 -3
.fz S 3 -0
```

will cause automatic reduction of font 3 by 3 points while the special characters would not be affected. Any ".fp" request specifying a font on some position must precede ".fz" requests relating to that position.

New Predefined Number Registers.

.k — Read-only. Contains the horizontal size of the text portion (without indent) of the current partially collected output line, if any, in the current environment.

.j — Read-only. A number representing the current adjustment mode and type. Can be saved and later given to the "ad" request to restore a previous mode.

.P — Read-only. 1 if the current page is being printed, and zero otherwise.

.L — Read-only. Contains the current line-spacing parameter ("ls").

c. — General register access to the input line-number in the current input file. Contains the same value as the read-only ".c" register.

Tbl — A Program to Format Tables

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Tbl is a document formatting preprocessor for *troff* or *nroff* which makes even fairly complex tables easy to specify and enter. It is available on the PDP-11 UNIX* system and on Honeywell 6000 GCOS. Tables are made up of columns which may be independently centered, right-adjusted, left-adjusted, or aligned by decimal points. Headings may be placed over single columns or groups of columns. A table entry may contain equations, or may consist of several rows of text. Horizontal or vertical lines may be drawn as desired in the table, and any table or element may be enclosed in a box. For example:

1970 Federal Budget Transfers
(in billions of dollars)

State	Taxes collected	Money spent	Net
New York	22.91	21.35	−1.56
New Jersey	8.33	6.96	−1.37
Connecticut	4.12	3.10	−1.02
Maine	0.74	0.67	−0.07
California	22.29	22.42	+0.13
New Mexico	0.70	1.49	+0.79
Georgia	3.30	4.28	+0.98
Mississippi	1.15	2.32	+1.17
Texas	9.33	11.13	+1.80

January 16, 1979

* UNIX is a Trademark/Service Mark of the Bell System

Tbl — A Program to Format Tables

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction.

Tbl turns a simple description of a table into a *troff* or *nroff* [1] program (list of commands) that prints the table. *Tbl* may be used on the PDP-11 UNIX [2] system and on the Honeywell 6000 GCOS system. It attempts to isolate a portion of a job that it can successfully handle and leave the remainder for other programs. Thus *tbl* may be used with the equation formatting program *eqn* [3] or various layout macro packages [4,5,6], but does not duplicate their functions.

This memorandum is divided into two parts. First we give the rules for preparing *tbl* input; then some examples are shown. The description of rules is precise but technical, and the beginning user may prefer to read the examples first, as they show some common table arrangements. A section explaining how to invoke *tbl* precedes the examples. To avoid repetition, henceforth read *troff* as "*troff* or *nroff*."

The input to *tbl* is text for a document, with tables preceded by a ".TS" (table start) command and followed by a ".TE" (table end) command. *Tbl* processes the tables, generating *troff* formatting commands, and leaves the remainder of the text unchanged. The ".TS" and ".TE" lines are copied, too, so that *troff* page layout macros (such as the memo formatting macros [4]) can use these lines to delimit and place tables as they see fit. In particular, any arguments on the ".TS" or ".TE" lines are copied but otherwise ignored, and may be used by document layout macro commands.

The format of the input is as follows:

```
text
.TS
table
.TE
text
.TS
table
.TE
text
. . .
```

where the format of each table is as follows:

```
.TS
options ;
format .
data
.TE
```

Each table is independent, and must contain formatting information followed by the data to be entered in the table. The formatting information, which describes the individual columns and rows of the table, may be preceded by a few options that affect the entire table. A detailed description of tables is given in the next section.

Input commands.

As indicated above, a table contains, first, global options, then a format section describing the layout of the table entries, and then the data to be printed. The format and data are always required, but not the options. The various parts of the table are entered as follows:

1) OPTIONS. There may be a single line of options affecting the whole table. If present, this line must follow the .TS line immediately and must contain a list of option names separated by spaces, tabs, or commas, and must be terminated by a semicolon. The allowable options are:

center — center the table (default is left-adjust);
expand — make the table as wide as the current line length;
box — enclose the table in a box;
allbox — enclose each item in the table in a box;
doublebox — enclose the table in two boxes;
tab (*x*) — use *x* instead of tab to separate data items.
linesize (*n*) — set lines or rules (e.g. from **box**) in *n* point type;
delim (*xy*) — recognize *x* and *y* as the *eqn* delimiters.

The *tbl* program tries to keep boxed tables on one page by issuing appropriate "need" (*.ne*) commands. These requests are calculated from the number of lines in the tables, and if there are spacing commands embedded in the input, these requests may be inaccurate; use normal *troff* procedures, such as keep-release macros, in that case. The user who must have a multi-page boxed table should use macros designed for this purpose, as explained below under 'Usage.'

2) FORMAT. The format section of the table specifies the layout of the columns. Each line in this section corresponds to one line of the table (except that the last line corresponds to all following lines up to the next .T&, if any — see below), and each line contains a key-letter for each column of the table. It is good practice to separate the key letters for each column by spaces or tabs. Each key-letter is one of the following:

L or l to indicate a left-adjusted column entry;
R or r to indicate a right-adjusted column entry;
C or c to indicate a centered column entry;
N or n to indicate a numerical column entry, to be aligned with other numerical entries so that the units digits of numbers line up;
A or a to indicate an alphabetic subcolumn; all corresponding entries are aligned on the left, and positioned so that the widest is centered within the column (see example on page 12);
S or s to indicate a spanned heading, i.e. to indicate that the entry from the previous column continues across this column (not allowed for the first column, obviously); or
^ to indicate a vertically spanned heading, i.e. to indicate that the entry from the previous row continues down through this row. (Not allowed for the first row of the table, obviously).

When numerical alignment is specified, a location for the decimal point is sought. The rightmost dot (.) adjacent to a digit is used as a decimal point; if there is no dot adjoining a digit, the rightmost digit is used as a units digit; if no alignment is indicated, the item is centered in the column. However, the special non-printing character string \& may be used to override unconditionally dots and digits, or to align alphabetic data; this string lines up where a dot normally would, and then disappears from the final output. In the example below, the items shown at the left will be aligned (in a numerical column) as

shown on the right:

```
13              13
4.2              4.2
26.4.12       26.4.12
abc               abc
abc\&           abc
43\&3.22        433.22
749.12         749.12
```

Note: If numerical data are used in the same column with wider **L** or **r** type table entries, the widest *number* is centered relative to the wider **L** or **r** items (**L** is used instead of **l** for readability; they have the same meaning as key-letters). Alignment within the numerical items is preserved. This is similar to the behavior of **a** type data, as explained above. However, alphabetic subcolumns (requested by the **a** key-letter) are always slightly indented relative to **L** items; if necessary, the column width is increased to force this. This is not true for **n** type entries.

Warning: the **n** and **a** items should not be used in the same column.

For readability, the key-letters describing each column should be separated by spaces. The end of the format section is indicated by a period. The layout of the key-letters in the format section resembles the layout of the actual data in the table. Thus a simple format might appear as:

```
c s s
l n n .
```

which specifies a table of three columns. The first line of the table contains a heading centered across all three columns; each remaining line contains a left-adjusted item in the first column followed by two columns of numerical data. A sample table in this format might be:

```
                Overall title
Item-a          34.22     9.1
Item-b          12.65      .02
Items: c,d,e    23         5.8
Total           69.87    14.92
```

There are some additional features of the key-letter system:

Horizontal lines — A key-letter may be replaced by '_' (underscore) to indicate a horizontal line in place of the corresponding column entry, or by '=' to indicate a double horizontal line. If an adjacent column contains a horizontal line, or if there are vertical lines adjoining this column, this horizontal line is extended to meet the nearby lines. If any data entry is provided for this column, it is ignored and a warning message is printed.

Vertical lines — A vertical bar may be placed between column key-letters. This will cause a vertical line between the corresponding columns of the table. A vertical bar to the left of the first key-letter or to the right of the last one produces a line at the edge of the table. If two vertical bars appear between key-letters, a double vertical line is drawn.

Space between columns — A number may follow the key-letter. This indicates the amount of separation between this column and the next column. The number normally specifies the separation in *ens* (one en is about the width of the letter 'n').* If the "expand" option is used, then these numbers are multiplied by a constant such that the table is as wide as the current line length. The default column separation

* More precisely, an en is a number of points (1 point = 1/72 inch) equal to half the current type size.

number is 3. If the separation is changed the worst case (largest space requested) governs.

Vertical spanning — Normally, vertically spanned items extending over several rows of the table are centered in their vertical range. If a key-letter is followed by **t** or **T**, any corresponding vertically spanned item will begin at the top line of its range.

Font changes — A key-letter may be followed by a string containing a font name or number preceded by the letter **f** or **F**. This indicates that the corresponding column should be in a different font from the default font (usually Roman). All font names are one or two letters; a one-letter font name should be separated from whatever follows by a space or tab. The single letters **B**, **b**, **I**, and **i** are shorter synonyms for **fB** and **fI**. Font change commands given with the table entries override these specifications.

Point size changes — A key-letter may be followed by the letter **p** or **P** and a number to indicate the point size of the corresponding table entries. The number may be a signed digit, in which case it is taken as an increment or decrement from the current point size. If both a point size and a column separation value are given, one or more blanks must separate them.

Vertical spacing changes — A key-letter may be followed by the letter **v** or **V** and a number to indicate the vertical line spacing to be used within a multi-line corresponding table entry. The number may be a signed digit, in which case it is taken as an increment or decrement from the current vertical spacing. A column separation value must be separated by blanks or some other specification from a vertical spacing request. This request has no effect unless the corresponding table entry is a text block (see below).

Column width indication — A key-letter may be followed by the letter **w** or **W** and a width value in parentheses. This width is used as a minimum column width. If the largest element in the column is not as wide as the width value given after the **w**, the largest element is assumed to be that wide. If the largest element in the column is wider than the specified value, its width is used. The width is also used as a default line length for included text blocks. Normal *troff* units can be used to scale the width value; if none are used, the default is ens. If the width specification is a unitless integer the parentheses may be omitted. If the width value is changed in a column, the *last* one given controls.

Equal width columns — A key-letter may be followed by the letter **e** or **E** to indicate equal width columns. All columns whose key-letters are followed by **e** or **E** are made the same width. This permits the user to get a group of regularly spaced columns.

Note: The order of the above features is immaterial; they need not be separated by spaces, except as indicated above to avoid ambiguities involving point size and font changes. Thus a numerical column entry in italic font and 12 point type with a minimum width of 2.5 inches and separated by 6 ens from the next column could be specified as

```
np12w(2.5i)fI 6
```

Alternative notation — Instead of listing the format of successive lines of a table on consecutive lines of the format section, successive line formats may be given on the same line, separated by commas, so that the format for the example above might have been written:

```
c s s, i n n .
```

Default — Column descriptors missing from the end of a format line are assumed to be **L**. The longest line in the format section, however, defines the number of columns in the table; extra columns in the data are ignored silently.

3) DATA. The data for the table are typed after the format. Normally, each table line is typed as one line of data. Very long input lines can be broken: any line whose last character is \ is combined with the following line (and the \ vanishes). The data for different columns (the table entries) are separated by tabs, or by whatever character has been specified in the option *tabs* option. There are a few special cases:

Troff commands within tables — An input line beginning with a '.' followed by anything but a number is assumed to be a command to *troff* and is passed through unchanged, retaining its position in the table. So, for example, space within a table may be produced by ".sp" commands in the data.

Full width horizontal lines — An input *line* containing only the character _ (underscore) or = (equal sign) is taken to be a single or double line, respectively, extending the full width of the *table.*

Single column horizontal lines — An input table *entry* containing only the character _ or = is taken to be a single or double line extending the full width of the *column.* Such lines are extended to meet horizontal or vertical lines adjoining this column. To obtain these characters explicitly in a column, either precede them by \& or follow them by a space before the usual tab or newline.

Short horizontal lines — An input table *entry* containing only the string _ is taken to be a single line as wide as the contents of the column. It is not extended to meet adjoining lines.

Repeated characters — An input table *entry* containing only a string of the form \R*x* where *x* is any character is replaced by repetitions of the character *x* as wide as the data in the column. The sequence of *x*'s is not extended to meet adjoining columns.

Vertically spanned items — An input table entry containing only the character string \^ indicates that the table entry immediately above spans downward over this row. It is equivalent to a table format key-letter of '^'.

Text blocks — In order to include a block of text as a table entry, precede it by T{ and follow it by T}. Thus the sequence

```
. . . T{
block of
text
T} . . .
```

is the way to enter, as a single entry in the table, something that cannot conveniently be typed as a simple string between tabs. Note that the T} end delimiter must begin a line; additional columns of data may follow after a tab on the same line. See the example on page 10 for an illustration of included text blocks in a table. If more than twenty or thirty text blocks are used in a table, various limits in the *troff* program are likely to be exceeded, producing diagnostics such as 'too many string/macro names' or 'too many number registers.'

Text blocks are pulled out from the table, processed separately by *troff*, and replaced in the table as a solid block. If no line length is specified in the *block of text* itself, or in the table format, the default is to use $L \times C/(N+1)$ where L is the current line length, C is the number of table columns spanned by the text, and N is the total number of columns in the table. The other parameters (point size, font, etc.) used in setting the *block of text* are those in effect at the beginning of the table (including the effect of the ".TS" macro) and any table format specifications of size, spacing and font, using the **p**, **v** and **f** modifiers to the column key-letters. Commands within the text block itself are also recognized, of course. However, *troff* commands within the table data but not within the text block do not affect that block.

Warnings: — Although any number of lines may be present in a table, only the first 200 lines are used in calculating the widths of the various columns. A multi-page table, of course, may be arranged as several single-page tables if this proves to be a problem. Other difficulties with formatting may arise because, in the calculation of column widths all table entries are assumed to be in the font and size being used when the ".TS" command was encountered, except for font and size changes indicated (a) in the table format section and (b) within the table data (as in the entry \s+3\fIdata\fP\s0). Therefore, although arbitrary *troff* requests may be sprinkled in a table, care must be taken to avoid confusing the width calculations; use requests such as '.ps' with care.

4) ADDITIONAL COMMAND LINES. If the format of a table must be changed after many similar lines, as with sub-headings or summarizations, the ".T&" (table continue) command can be used to change column parameters. The outline of such a table input is:

```
.TS
options ;
format .
data
. . .
.T&
format .
data
.T&
format .
data
.TE
```

as in the examples on pages 10 and 12. Using this procedure, each table line can be close to its corresponding format line.

Warning: it is not possible to change the number of columns, the space between columns, the global options such as *box*, or the selection of columns to be made equal width.

Usage.

On UNIX, *tbl* can be run on a simple table with the command

```
tbl input-file | troff
```

but for more complicated use, where there are several input files, and they contain equations and *ms* memorandum layout commands as well as tables, the normal command would be

```
tbl file-1 file-2 . . . | eqn | troff -ms
```

and, of course, the usual options may be used on the *troff* and *eqn* commands. The usage for *nroff* is similar to that for *troff*, but only TELETYPE® Model 37 and Diablo-mechanism (DASI or GSI) terminals can print boxed tables directly.

For the convenience of users employing line printers without adequate driving tables or post-filters, there is a special *-TX* command line option to *tbl* which produces output that does not have fractional line motions in it. The only other command line options recognized by *tbl* are *-ms* and *-mm* which are turned into commands to fetch the corresponding macro files; usually it is more convenient to place these arguments on the *troff* part of the command line, but they are accepted by *tbl* as well.

Note that when *eqn* and *tbl* are used together on the same file *tbl* should be used first. If there are no equations within tables, either order works, but it is usually faster to run *tbl* first, since *eqn* normally produces a larger expansion of the input than *tbl*. However, if there are equations within tables (using the *delim* mechanism in *eqn*), *tbl* must be first or the output will be scrambled. Users must also beware of using equations in **n**-style columns; this is nearly

always wrong, since *tbl* attempts to split numerical format items into two parts and this is not possible with equations. The user can defend against this by giving the *delim(xx)* table option; this prevents splitting of numerical columns within the delimiters. For example, if the *eqn* delimiters are *$$*, giving *delim($$)* a numerical column such as "1245 $+- 16$" will be divided after 1245, not after 16.

Tbl limits tables to twenty columns; however, use of more than 16 numerical columns may fail because of limits in *troff*, producing the 'too many number registers' message. *Troff* number registers used by *tbl* must be avoided by the user within tables; these include two-digit names from 31 to 99, and names of the forms *#x*, *x+*, *x|*, *-x*, and *x-*, where *x* is any lower case letter. The names *##*, *#-*, and *#^* are also used in certain circumstances. To conserve number register names, the **n** and **a** formats share a register; hence the restriction above that they may not be used in the same column.

For aid in writing layout macros, *tbl* defines a number register TW which is the table width; it is defined by the time that the ".TE" macro is invoked and may be used in the expansion of that macro. More importantly, to assist in laying out multi-page boxed tables the macro T# is defined to produce the bottom lines and side lines of a boxed table, and then invoked at its end. By use of this macro in the page footer a multi-page table can be boxed. In particular, the *ms* macros can be used to print a multi-page boxed table with a repeated heading by giving the argument H to the ".TS" macro. If the table start macro is written

```
.TS H
```

a line of the form

```
.TH
```

must be given in the table after any table heading (or at the start if none). Material up to the ".TH" is placed at the top of each page of table; the remaining lines in the table are placed on several pages as required. Note that this is *not* a feature of *tbl*, but of the *ms* layout macros.

Examples.

Here are some examples illustrating features of *tbl*. The symbol ⇥ in the input represents a tab character.

Input:

```
.TS
box;
c c c
l l l.
Language⇥Authors⇥Runs on

Fortran⇥Many⇥Almost anything
PL/1⇥IBM⇥360/370
C⇥BTL⇥11/45,H6000,370
BLISS⇥Carnegie-Mellon⇥PDP-10,11
IDS⇥Honeywell⇥H6000
Pascal⇥Stanford⇥370
.TE
```

Output:

Language	Authors	Runs on
Fortran	Many	Almost anything
PL/1	IBM	360/370
C	BTL	11/45,H6000,370
BLISS	Carnegie-Mellon	PDP-10,11
IDS	Honeywell	H6000
Pascal	Stanford	370

Input:

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

Output:

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

* (first quarter only)

Input:

```
.TS
box;
c s s
c|c|c
|||n.
Major New York Bridges
=
Bridge Ⓣ Designer Ⓣ Length

Brooklyn Ⓣ J. A. Roebling Ⓣ 1595
Manhattan Ⓣ G. Lindenthal Ⓣ 1470
Williamsburg Ⓣ L. L. Buck Ⓣ 1600

Queensborough Ⓣ Palmer & Ⓣ 1182
Ⓣ Hornbostel

Ⓣ Ⓣ 1380
Triborough Ⓣ O. H. Ammann Ⓣ_
Ⓣ Ⓣ 383

Bronx Whitestone Ⓣ O. H. Ammann Ⓣ 2300
Throgs Neck Ⓣ O. H. Ammann Ⓣ 1800

George Washington Ⓣ O. H. Ammann Ⓣ 3500
.TE
```

Output:

Major New York Bridges		
Bridge	Designer	Length
Brooklyn	J. A. Roebling	1595
Manhattan	G. Lindenthal	1470
Williamsburg	L. L. Buck	1600
Queensborough	Palmer & Hornbostel	1182
Triborough	O. H. Ammann	1380
		383
Bronx Whitestone	O. H. Ammann	2300
Throgs Neck	O. H. Ammann	1800
George Washington	O. H. Ammann	3500

Input:

```
.TS
c c
np-2 | n | .
T{Stack
T}_
1T}46
T}_
2T}23
T}_
3T}15
T}_
4T}6.5
T}_
5T}2.1
T}_
.TE
```

Output:

	Stack
1	46
2	23
3	15
4	6.5
5	2.1

Input:

```
.TS
box;
L L L
L L _
L L | LB
L L _
L L L.
january T{february T}march
april T{may
june T{july T}Months
august T{september
october T{november T}december
.TE
```

Output:

january	february	march
april	may	**Months**
june	july	
august	september	
october	november	december

Input:

```
.TS
box;
cfB s s s.
Composition of Foods

_
.T&
c |c s s
c |c s s
c |c |c |c.
Food ⓉPercent by Weight
\^Ⓣ_
\^ⓉProtein Ⓣ Fat ⓉCarbo-
\^Ⓣ\^Ⓣ\^Ⓣhydrate

_
.T&
l |n |n |n.
Apples Ⓣ.4Ⓣ.5Ⓣ13.0
Halibut Ⓣ18.4Ⓣ5.2Ⓣ. . .
Lima beans Ⓣ7.5Ⓣ.8Ⓣ22.0
Milk Ⓣ3.3Ⓣ4.0Ⓣ5.0
Mushrooms Ⓣ3.5Ⓣ.4Ⓣ6.0
Rye bread Ⓣ9.0Ⓣ.6Ⓣ52.7
.TE
```

Output:

Composition of Foods			
Food	Percent by Weight		
	Protein	Fat	Carbo-hydrate
Apples	.4	.5	13.0
Halibut	18.4	5.2	...
Lima beans	7.5	.8	22.0
Milk	3.3	4.0	5.0
Mushrooms	3.5	.4	6.0
Rye bread	9.0	.6	52.7

Input:

```
.TS
allbox;
cfI s  s
c  cw(1i)  cw(1i)
lp9 lp9 lp9.
New York Area Rocks
Era Ⓣ Formation Ⓣ Age (years)
Precambrian Ⓣ Reading Prong Ⓣ > 1 billion
Paleozoic Ⓣ Manhattan Prong Ⓣ 400 million
Mesozoic Ⓣ T{
.na
Newark Basin, incl.
Stockton, Lockatong, and Brunswick
formations; also Watchungs
and Palisades.
T} Ⓣ 200 million
Cenozoic Ⓣ Coastal Plain Ⓣ T{
On Long Island 30,000 years:
Cretaceous sediments redeposited
by recent glaciation.
.ad
T}
.TE
```

Output:

New York Area Rocks		
Era	Formation	Age (years)
Precambrian	Reading Prong	> 1 billion
Paleozoic	Manhattan Prong	400 million
Mesozoic	Newark Basin, incl. Stockton, Lockatong, and Brunswick formations; also Watchungs and Palisades.	200 million
Cenozoic	Coastal Plain	On Long Island 30,000 years. Cretaceous sediments redeposited by recent glaciation.

Input:

```
.EQ
delim $$
.EN

. . .

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

Output:

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

Input:

```
.TS
box, tab(:);
cb s s s s
cp-2 s s s s
c|c|c|c|c
c|c|c|c|c
r2|n2|n2|n2|n.
Readability of Text
Line Width and Leading for 10-Point Type
=
Line:Set:1-Point:2-Point:4-Point
Width:Solid:Leading:Leading:Leading

9 Pica:\-9.3:\-6.0:\-5.3:\-7.1
14 Pica:\-4.5:\-0.6:\-0.3:\-1.7
19 Pica:\-5.0:\-5.1: 0.0:\-2.0
31 Pica:\-3.7:\-3.8:\-2.4:\-3.6
43 Pica:\-9.1:\-9.0:\-5.9:\-8.8
.TE
```

Output:

Readability of Text

Line Width and Leading for 10-Point Type

Line Width	Set Solid	1-Point Leading	2-Point Leading	4-Point Leading
9 Pica	−9.3	−6.0	−5.3	−7.1
14 Pica	−4.5	−0.6	−0.3	−1.7
19 Pica	−5.0	−5.1	0.0	−2.0
31 Pica	−3.7	−3.8	−2.4	−3.6
43 Pica	−9.1	−9.0	−5.9	−8.8

Input:

```
.TS
c s
cip-2 s
l n
a n.
Some London Transport Statistics
(Year 1964)
Railway route miles ⊜ 244
Tube ⊜ 66
Sub-surface ⊜ 22
Surface ⊜ 156
.sp .5
.T&
l r
a r.
Passenger traffic \- railway
Journeys ⊜ 674 million
Average length ⊜ 4.55 miles
Passenger miles ⊜ 3,066 million
.T&
l r
a r.
Passenger traffic \- road
Journeys ⊜ 2,252 million
Average length ⊜ 2.26 miles
Passenger miles ⊜ 5,094 million
.T&
l n
a n.
.sp .5
Vehicles ⊜ 12,521
Railway motor cars ⊜ 2,905
Railway trailer cars ⊜ 1,269
Total railway ⊜ 4,174
Omnibuses ⊜ 8,347
.T&
l n
a n.
.sp .5
Staff ⊜ 73,739
Administrative, etc. ⊜ 5,582
Civil engineering ⊜ 5,134
Electrical eng. ⊜ 1,714
Mech. eng. \- railway ⊜ 4,310
Mech. eng. \- road ⊜ 9,152
Railway operations ⊜ 8,930
Road operations ⊜ 35,946
Other ⊜ 2,971
.TE
```

Output:

Some London Transport Statistics

(Year 1964)

Railway route miles	244
Tube	66
Sub-surface	22
Surface	156
Passenger traffic – railway	
Journeys	674 million
Average length	4.55 miles
Passenger miles	3,066 million
Passenger traffic – road	
Journeys	2,252 million
Average length	2.26 miles
Passenger miles	5,094 million
Vehicles	12,521
Railway motor cars	2,905
Railway trailer cars	1,269
Total railway	4,174
Omnibuses	8,347
Staff	73,739
Administrative, etc.	5,582
Civil engineering	5,134
Electrical eng.	1,714
Mech. eng. – railway	4,310
Mech. eng. – road	9,152
Railway operations	8,930
Road operations	35,946
Other	2,971

Input:

```
.ps 8
.vs 10p
.TS
center box;
c s s
ci s s
c c c
lB l n.
New Jersey Representatives
(Democrats)
.sp .5
Name ⓣ Office address ⓣ Phone
.sp .5
James J. Florio ⓣ 23 S. White Horse Pike, Somerdale 08083 ⓣ 609-627-8222
William J. Hughes ⓣ 2920 Atlantic Ave., Atlantic City 08401 ⓣ 609-345-4844
James J. Howard ⓣ 801 Bangs Ave., Asbury Park 07712 ⓣ 201-774-1600
Frank Thompson, Jr. ⓣ 10 Rutgers Pl., Trenton 08618 ⓣ 609-599-1619
Andrew Maguire ⓣ 115 W. Passaic St., Rochelle Park 07662 ⓣ 201-843-0240
Robert A. Roe ⓣ U.S.P.O., 194 Ward St., Paterson 07510 ⓣ 201-523-5152
Henry Helstoski ⓣ 666 Paterson Ave., East Rutherford 07073 ⓣ 201-939-9090
Peter W. Rodino, Jr. ⓣ Suite 1435A, 970 Broad St., Newark 07102 ⓣ 201-645-3213
Joseph G. Minish ⓣ 308 Main St., Orange 07050 ⓣ 201-645-6363
Helen S. Meyner ⓣ 32 Bridge St., Lambertville 08530 ⓣ 609-397-1830
Dominick V. Daniels ⓣ 895 Bergen Ave., Jersey City 07306 ⓣ 201-659-7700
Edward J. Patten ⓣ Natl. Bank Bldg., Perth Amboy 08861 ⓣ 201-826-4610
.sp .5
.T&
ci s s
lB l n.
(Republicans)
.sp .5v
Millicent Fenwick ⓣ 41 N. Bridge St., Somerville 08876 ⓣ 201-722-8200
Edwin B. Forsythe ⓣ 301 Mill St., Moorestown 08057 ⓣ 609-235-6622
Matthew J. Rinaldo ⓣ 1961 Morris Ave., Union 07083 ⓣ 201-687-4235
.TE
.ps 10
.vs 12p
```

Output:

New Jersey Representatives

Name	Office address	Phone
	(Democrats)	
James J. Florio	23 S. White Horse Pike, Somerdale 08083	609-627-8222
William J. Hughes	2920 Atlantic Ave., Atlantic City 08401	609-345-4844
James J. Howard	801 Bangs Ave., Asbury Park 07712	201-774-1600
Frank Thompson, Jr.	10 Rutgers Pl., Trenton 08618	609-599-1619
Andrew Maguire	115 W. Passaic St., Rochelle Park 07662	201-843-0240
Robert A. Roe	U.S.P.O., 194 Ward St., Paterson 07510	201-523-5152
Henry Helstoski	666 Paterson Ave., East Rutherford 07073	201-939-9090
Peter W. Rodino, Jr.	Suite 1435A, 970 Broad St., Newark 07102	201-645-3213
Joseph G. Minish	308 Main St., Orange 07050	201-645-6363
Helen S. Meyner	32 Bridge St., Lambertville 08530	609-397-1830
Dominick V. Daniels	895 Bergen Ave., Jersey City 07306	201-659-7700
Edward J. Patten	Natl. Bank Bldg., Perth Amboy 08861	201-826-4610
	(Republicans)	
Millicent Fenwick	41 N Bridge St., Somerville 08876	201-722-8200
Edwin B. Forsythe	301 Mill St., Moorestown 08057	609-235-6622
Matthew J. Rinaldo	1961 Morris Ave., Union 07083	201-687-4235

This is a paragraph of normal text placed here only to indicate where the left and right margins are. In this way the reader can judge the appearance of centered tables or expanded tables, and observe how such tables are formatted.

Input:

```
.TS
expand;
c s s s
c c c c
l l n n.
Bell Labs Locations
Name ⓣ Address ⓣ Area Code ⓣ Phone
Holmdel ⓣ Holmdel, N. J. 07733 ⓣ 201 ⓣ 949-3000
Murray Hill ⓣ Murray Hill, N. J. 07974 ⓣ 201 ⓣ 582-6377
Whippany ⓣ Whippany, N. J. 07981 ⓣ 201 ⓣ 386-3000
Indian Hill ⓣ Naperville, Illinois 60540 ⓣ 312 ⓣ 690-2000
.TE
```

Output:

Bell Labs Locations

Name	Address	Area Code	Phone
Holmdel	Holmdel, N. J. 07733	201	949-3000
Murray Hill	Murray Hill, N. J. 07974	201	582-6377
Whippany	Whippany, N. J. 07981	201	386-3000
Indian Hill	Naperville, Illinois 60540	312	690-2000

Input:

```
.TS
box;
cb  s  s  s
c|c|c  s
ltiw(1i) | ltw(2i) | lp8 | lw(1.6i)p8.
Some Interesting Places

_
Name⓽Description⓽Practical Information

_
T{
American Museum of Natural History
T}⓽T{
The collections fill 11.5 acres (Michelin) or 25 acres (MTA)
of exhibition halls on four floors.  There is a full-sized replica
of a blue whale and the world's largest star sapphire (stolen in 1964).
T}⓽Hours⓽10-5, ex. Sun 11-5, Wed. to 9
\^⓽\^⓽Location⓽T{
Central Park West & 79th St.
T}
\^⓽\^⓽Admission⓽Donation: $1.00 asked
\^⓽\^⓽Subway⓽AA to 81st St.
\^⓽\^⓽Telephone⓽212-873-4225

_
Bronx Zoo⓽T{
About a mile long and .6 mile wide, this is the largest zoo in America.
A lion eats 18 pounds
of meat a day while a sea lion eats 15 pounds of fish.
T}⓽Hours⓽T{
10-4:30 winter, to 5:00 summer
T}
\^⓽\^⓽Location⓽T{
185th St. & Southern Blvd, the Bronx.
T}
\^⓽\^⓽Admission⓽$1.00, but Tu,We,Th free
\^⓽\^⓽Subway⓽2, 5 to East Tremont Ave.
\^⓽\^⓽Telephone⓽212-933-1759

_
Brooklyn Museum⓽T{
Five floors of galleries contain American and ancient art.
There are American period rooms and architectural ornaments saved
from wreckers, such as a classical figure from Pennsylvania Station.
T}⓽Hours⓽Wed-Sat, 10-5, Sun 12-5
\^⓽\^⓽Location⓽T{
Eastern Parkway & Washington Ave., Brooklyn.
T}
\^⓽\^⓽Admission⓽Free
\^⓽\^⓽Subway⓽2,3 to Eastern Parkway.
\^⓽\^⓽Telephone⓽212-638-5000

_
T{
New-York Historical Society
T}⓽T{
All the original paintings for Audubon's
.I
Birds of America
.R
are here, as are exhibits of American decorative arts, New York history,
Hudson River school paintings, carriages, and glass paperweights.
T}⓽Hours⓽T{
Tues-Fri & Sun, 1-5; Sat 10-5
T}
\^⓽\^⓽Location⓽T{
Central Park West & 77th St.
T}
\^⓽\^⓽Admission⓽Free
\^⓽\^⓽Subway⓽AA to 81st St.
\^⓽\^⓽Telephone⓽212-873-3400
.TE
```

Output:

Some Interesting Places			
Name	Description	Practical Information	
American Museum of Natural History	The collections fill 11.5 acres (Michelin) or 25 acres (MTA) of exhibition halls on four floors. There is a full-sized replica of a blue whale and the world's largest star sapphire (stolen in 1964).	Hours Location Admission Subway Telephone	10-5, ex. Sun 11-5, Wed. to 9 Central Park West & 79th St. Donation $1.00 asked AA to 81st St. 212-873-4225
Bronx Zoo	About a mile long and .6 mile wide, this is the largest zoo in America. A lion eats 18 pounds of meat a day while a sea lion eats 15 pounds of fish.	Hours Location Admission Subway Telephone	10-4:30 winter, to 5:00 summer 185th St. & Southern Blvd, the Bronx. $1.00, but Tu,We,Th free 2, 5 to East Tremont Ave. 212-933-1759
Brooklyn Museum	Five floors of galleries contain American and ancient art. There are American period rooms and architectural ornaments saved from wreckers, such as a classical figure from Pennsylvania Station.	Hours Location Admission Subway Telephone	Wed-Sat, 10-5, Sun 12-5 Eastern Parkway & Washington Ave., Brooklyn. Free 2,3 to Eastern Parkway. 212-638-5000
New-York Historical Society	All the original paintings for Audubon's *Birds of America* are here, as are exhibits of American decorative arts, New York history, Hudson River school paintings, carriages, and glass paperweights.	Hours Location Admission Subway Telephone	Tues-Fri & Sun, 1-5; Sat 10-5 Central Park West & 77th St. Free AA to 81st St. 212-873-3400

Acknowledgments.

Many thanks are due to J. C. Blinn, who has done a large amount of testing and assisted with the design of the program. He has also written many of the more intelligible sentences in this document and helped edit all of it. All phototypesetting programs on UNIX are dependent on the work of the late J. F. Ossanna, whose assistance with this program in particular had been most helpful. This program is patterned on a table formatter originally written by J. F. Gimpel. The assistance of T. A. Dolotta, B. W. Kernighan, and J. N. Sturman is gratefully acknowledged.

References.

[1] J. F. Ossanna, *NROFF/TROFF User's Manual.* Computing Science Technical Report No. 54, Bell Laboratories, 1976.

[2] K. Thompson and D. M. Ritchie, "The UNIX Time-Sharing System," *Comm. ACM.* 17, pp. 365–75 (1974).

[3] B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. ACM.* 18, pp. 151–57 (1975).

[4] M. E. Lesk, *Typing Documents on UNIX.* UNIX Programmer's Manual, Volume 2.

[5] M. E. Lesk and B. W. Kernighan, *Computer Typesetting of Technical Journals on UNIX, Proc. AFIPS NCC*, vol. 46, pp. 879-888 (1977).

[6] J. R. Mashey and D. W. Smith, "Documentation Tools and Techniques," *Proc. 2nd Int. Conf. on Software Engineering*, pp. 177-181 (October, 1976).

List of Tbl Command Characters and Words

Command	*Meaning*	*Section*
a A	Alphabetic subcolumn	2
allbox	Draw box around all items	1
b B	Boldface item	2
box	Draw box around table	1
c C	Centered column	2
center	Center table in page	1
doublebox	Doubled box around table	1
e E	Equal width columns	2
expand	Make table full line width	1
f F	Font change	2
i I	Italic item	2
l L	Left adjusted column	2
n N	Numerical column	2
nnn	Column separation	2
p P	Point size change	2
r R	Right adjusted column	2
s S	Spanned item	2
t T	Vertical spanning at top	2
tab (*x*)	Change data separator character	1
T{ T}	Text block	3
v V	Vertical spacing change	2
w W	Minimum width value	2
.xx	Included *troff* command	3
|	Vertical line	2
||	Double vertical line	2
^	Vertical span	2
\^	Vertical span	3
=	Double horizontal line	2,3
_	Horizontal line	2,3
_	Short horizontal line	3
\Rx	Repeat character	3

A System for Typesetting Mathematics

Brian W. Kernighan and Lorinda L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes the design and implementation of a system for typesetting mathematics. The language has been designed to be easy to learn and to use by people (for example, secretaries and mathematical typists) who know neither mathematics nor typesetting. Experience indicates that the language can be learned in an hour or so, for it has few rules and fewer exceptions. For typical expressions, the size and font changes, positioning, line drawing, and the like necessary to print according to mathematical conventions are all done automatically. For example, the input

sum from i=0 to infinity x sub i = pi over 2

produces

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

The syntax of the language is specified by a small context-free grammar; a compiler-compiler is used to make a compiler that translates this language into typesetting commands. Output may be produced on either a phototypesetter or on a terminal with forward and reverse half-line motions. The system interfaces directly with text formatting programs, so mixtures of text and mathematics may be handled simply.

This paper is a revision of a paper originally published in CACM, March, 1975.

1. Introduction

"Mathematics is known in the trade as *difficult*, or *penalty*, *copy* because it is slower, more difficult, and more expensive to set in type than any other kind of copy normally occurring in books and journals." [1]

One difficulty with mathematical text is the multiplicity of characters, sizes, and fonts. An expression such as

$$\lim_{x \to \pi/2} (\tan x)^{\sin 2x} = 1$$

requires an intimate mixture of roman, italic and greek letters, in three sizes, and a special character or two. ("Requires" is perhaps the wrong word, but mathematics has its own typographical conventions which are quite different from those of ordinary text.) Typesetting such an expression by traditional methods is still an essentially manual operation.

A second difficulty is the two dimensional character of mathematics, which the superscript and limits in the preceding example showed in its simplest form. This is carried further by

$$a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \cdots}}}$$

and still further by

$$\int \frac{dx}{ae^{mx} - be^{-mx}} = \begin{cases} \dfrac{1}{2m\sqrt{ab}} \log \dfrac{\sqrt{a}e^{mx} - \sqrt{b}}{\sqrt{a}e^{mx} + \sqrt{b}} \\ \dfrac{1}{m\sqrt{ab}} \tanh^{-1}\left(\dfrac{\sqrt{a}}{\sqrt{b}} e^{mx}\right) \\ \dfrac{-1}{m\sqrt{ab}} \coth^{-1}\left(\dfrac{\sqrt{a}}{\sqrt{b}} e^{mx}\right) \end{cases}$$

These examples also show line-drawing, built-up characters like braces and radicals, and a spectrum of positioning problems. (Section 6 shows

what a user has to type to produce these on our system.)

2. Photocomposition

Photocomposition techniques can be used to solve some of the problems of typesetting mathematics. A phototypesetter is a device which exposes a piece of photographic paper or film, placing characters wherever they are wanted. The Graphic Systems phototypesetter[2] on the UNIX operating system[3] works by shining light through a character stencil. The character is made the right size by lenses, and the light beam directed by fiber optics to the desired place on a piece of photographic paper. The exposed paper is developed and typically used in some form of photo-offset reproduction.

On UNIX, the phototypesetter is driven by a formatting program called TROFF [4]. TROFF was designed for setting running text. It also provides all of the facilities that one needs for doing mathematics, such as arbitrary horizontal and vertical motions, line-drawing, size changing, but the syntax for describing these special operations is difficult to learn, and difficult even for experienced users to type correctly.

For this reason we decided to use TROFF as an "assembly language," by designing a language for describing mathematical expressions, and compiling it into TROFF.

3. Language Design

The fundamental principle upon which we based our language design is that the language should be easy to use by people (for example, secretaries) who know neither mathematics nor typesetting.

This principle implies several things. First, "normal" mathematical conventions about operator precedence, parentheses, and the like cannot be used, for to give special meaning to such characters means that the user has to understand what he or she is typing. Thus the language should not assume, for instance, that parentheses are always balanced, for they are not in the half-open interval $(a,b]$. Nor should it assume that that $\sqrt{a+b}$ can be replaced by $(a+b)^{.5}$, or that $1/(1-x)$ is better written as $\frac{1}{1-x}$ (or vice versa).

Second, there should be relatively few rules, keywords, special symbols and operators, and the like. This keeps the language easy to learn and remember. Furthermore, there should be few exceptions to the rules that do exist: if something works in one situation, it should work everywhere. If a variable can have a subscript, then a subscript can have a subscript, and so on without limit.

Third, "standard" things should happen automatically. Someone who types "x=y+z+1" should get "$x=y+z+1$". Subscripts and superscripts should automatically be printed in an appropriately smaller size, with no special intervention. Fraction bars have to be made the right length and positioned at the right height. And so on. Indeed a mechanism for overriding default actions has to exist, but its application is the exception, not the rule.

We assume that the typist has a reasonable picture (a two-dimensional representation) of the desired final form, as might be handwritten by the author of a paper. We also assume that the input is typed on a computer terminal much like an ordinary typewriter. This implies an input alphabet of perhaps 100 characters, none of them special.

A secondary, but still important, goal in our design was that the system should be easy to implement, since neither of the authors had any desire to make a long-term project of it. Since our design was not firm, it was also necessary that the program be easy to change at any time.

To make the program easy to build and to change, and to guarantee regularity ("it should work everywhere"), the language is defined by a context-free grammar, described in Section 5. The compiler for the language was built using a compiler-compiler.

A priori, the grammar/compiler-compiler approach seemed the right thing to do. Our subsequent experience leads us to believe that any other course would have been folly. The original language was designed in a few days. Construction of a working system sufficient to try significant examples required perhaps a person-month. Since then, we have spent a modest amount of additional time over several years tuning, adding facilities, and occasionally changing the language as users make criticisms and suggestions.

We also decided quite early that we would let TROFF do our work for us whenever possible. TROFF is quite a powerful program, with a macro facility, text and arithmetic variables, numerical computation and testing, and conditional branching. Thus we have been able to avoid writing a lot of mundane but tricky software. For example, we store no text strings, but simply pass them on to TROFF. Thus we avoid having to write a storage management package. Furthermore, we have been able to isolate ourselves from most details of the particular device and character set currently in use. For example, we let TROFF compute the widths of all strings of

characters; we need know nothing about them.

A third design goal is special to our environment. Since our program is only useful for typesetting mathematics, it is necessary that it interface cleanly with the underlying typesetting language for the benefit of users who want to set intermingled mathematics and text (the usual case). The standard mode of operation is that when a document is typed, mathematical expressions are input as part of the text, but marked by user settable delimiters. The program reads this input and treats as comments those things which are not mathematics, simply passing them through untouched. At the same time it converts the mathematical input into the necessary TROFF commands. The resulting ioutput is passed directly to TROFF where the comments and the mathematical parts both become text and/or TROFF commands.

4. The Language

We will not try to describe the language precisely here; interested readers may refer to the appendix for more details. Throughout this section, we will write expressions exactly as they are handed to the typesetting program (hereinafter called "EQN"), except that we won't show the delimiters that the user types to mark the beginning and end of the expression. The interface between EQN and TROFF is described at the end of this section.

As we said, typing `x=y+z+1` should produce $x=y+z+1$, and indeed it does. Variables are made italic, operators and digits become roman, and normal spacings between letters and operators are altered slightly to give a more pleasing appearance.

Input is free-form. Spaces and new lines in the input are used by EQN to separate pieces of the input; they are not used to create space in the output. Thus

```
x
=
y
+ z + 1
```

also gives $x=y+z+1$. Free-form input is easier to type initially; subsequent editing is also easier, for an expression may be typed as many short lines.

Extra white space can be forced into the output by several characters of various sizes. A tilde "~" gives a space equal to the normal word spacing in text; a circumflex gives half this much, and a tab charcter spaces to the next tab stop.

Spaces (or tildes, etc.) also serve to delimit pieces of the input. For example, to get

$$f(t)=2\pi\int \sin(\omega t)dt$$

we write

```
f(t) = 2 pi int sin ( omega t )dt
```

Here spaces are *necessary* in the input to indicate that *sin*, *pi*, *int*, and *omega* are special, and potentially worth special treatment. EQN looks up each such string of characters in a table, and if appropriate gives it a translation. In this case, *pi* and *omega* become their greek equivalents, *int* becomes the integral sign (which must be moved down and enlarged so it looks "right"), and *sin* is made roman, following conventional mathematical practice. Parentheses, digits and operators are automatically made roman wherever found.

Fractions are specified with the keyword *over*:

```
a+b over c+d+e = 1
```

produces

$$\frac{a+b}{c+d+e}=1$$

Similarly, subscripts and superscripts are introduced by the keywords *sub* and *sup*:

$$x^2+y^2=z^2$$

is produced by

```
x sup 2 + y sup 2 = z sup 2
```

The spaces after the 2's are necessary to mark the end of the superscripts; similarly the keyword *sup* has to be marked off by spaces or some equivalent delimiter. The return to the proper baseline is automatic. Multiple levels of subscripts or superscripts are of course allowed: "x sup y sup z" is x^{y^z}. The construct "something *sub* something *sup* something" is recognized as a special case, so "x sub i sup 2" is x_i^2 instead of x_i^2.

More complicated expressions can now be formed with these primitives:

$$\frac{\partial^2 f}{\partial x^2}=\frac{x^2}{a^2}+\frac{y^2}{b^2}$$

is produced by

```
{partial sup 2 f} over {partial x sup 2} =
x sup 2 over a sup 2 + y sup 2 over b sup 2
```

Braces {} are used to group objects together; in this case they indicate unambiguously what goes over what on the left-hand side of the expression. The language defines the precedence of *sup* to be higher than that of *over*, so no braces are needed to get the correct association on the right side. Braces can always be used when in doubt about precedence.

The braces convention is an example of

the power of using a recursive grammar to define the language. It is part of the language that if a construct can appear in some context, then *any expression* in braces can also occur in that context.

There is a *sqrt* operator for making square roots of the appropriate size: "sqrt a+b" produces $\sqrt{a+b}$, and

```
x = {-b +- sqrt{b sup 2 -4ac}} over 2a
```

is

$$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

Since large radicals look poor on our typesetter, *sqrt* is not useful for tall expressions.

Limits on summations, integrals and similar constructions are specified with the keywords *from* and *to*. To get

$$\sum_{i=0}^{\infty} x_i = 0$$

we need only type

```
sum from i=0 to inf x sub i -> 0
```

Centering and making the Σ big enough and the limits smaller are all automatic. The *from* and *to* parts are both optional, and the central part (e.g., the Σ) can in fact be anything:

```
lim from {x -> pi /2} ( tan~x) = inf
```

is

$$\lim_{x \to \pi/2} (\tan x) = \infty$$

Again, the braces indicate just what goes into the *from* part.

There is a facility for making braces, brackets, parentheses, and vertical bars of the right height, using the keywords *left* and *right*:

```
left [ x+y over 2a right ]~=~1
```

makes

$$\left[\frac{x+y}{2a} \right] = 1$$

A *left* need not have a corresponding *right*, as we shall see in the next example. Any characters may follow *left* and *right*, but generally only various parentheses and bars are meaningful.

Big brackets, etc., are often used with another facility, called *piles*, which make vertical piles of objects. For example, to get

$$sign(x) \equiv \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

we can type

```
sign (x) ~=~ left {
   rpile {1 above 0 above -1}
   ~lpile {if above if above if}
   ~lpile {x>0 above x=0 above x<0}
```

The construction "left {" makes a left brace big enough to enclose the "rpile {...}", which is a right-justified pile of "above ... above ...". "lpile" makes a left-justified pile. There are also centered piles. Because of the recursive language definition, a pile can contain any number of elements; any element of a pile can of course contain piles.

Although EQN makes a valiant attempt to use the right sizes and fonts, there are times when the default assumptions are simply not what is wanted. For instance the italic *sign* in the previous example would conventionally be in roman. Slides and transparencies often require larger characters than normal text. Thus we also provide size and font changing commands: "size 12 bold {A~x~=~y}" will produce $\mathbf{A\ x = y}$. *Size* is followed by a number representing a character size in points. (One point is 1/72 inch; this paper is set in 9 point type.)

If necessary, an input string can be quoted in "...", which turns off grammatical significance, and any font or spacing changes that might otherwise be done on it. Thus we can say

```
lim~ roman "sup" ~x sub n = 0
```

to ensure that the supremum doesn't become a superscript:

$$\lim \sup x_n = 0$$

Diacritical marks, long a problem in traditional typesetting, are straightforward:

$$\underline{\dot{x}} + \hat{x} + \tilde{y} + \hat{X} + \ddot{Y} = \overline{z + Z}$$

is made by typing

```
x dot under + x hat + y tilde
+ X hat + Y dotdot = z+Z bar
```

There are also facilities for globally changing default sizes and fonts, for example for making viewgraphs or for setting chemical equations. The language allows for matrices, and for lining up equations at the same horizontal position.

Finally, there is a definition facility, so a user can say

```
define name "..."
```

at any time in the document; henceforth, any occurrence of the token "name" in an expression will be expanded into whatever was inside the double quotes in its definition. This lets users tailor the language to their own

specifications, for it is quite possible to redefine keywords like *sup* or *over*. Section 6 shows an example of definitions.

The EQN preprocessor reads intermixed text and equations, and passes its output to TROFF. Since TROFF uses lines beginning with a period as control words (e.g., ".ce" means "center the next output line"), EQN uses the sequence ".EQ" to mark the beginning of an equation and ".EN" to mark the end. The ".EQ" and ".EN" are passed through to TROFF untouched, so they can also be used by a knowledgeable user to center equations, number them automatically, etc. By default, however, ".EQ" and ".EN" are simply ignored by TROFF, so by default equations are printed in-line.

".EQ" and ".EN" can be supplemented by TROFF commands as desired; for example, a centered display equation can be produced with the input:

```
.ce
.EQ
x sub i = y sub i ...
.EN
```

Since it is tedious to type ".EQ" and ".EN" around very short expressions (single letters, for instance), the user can also define two characters to serve as the left and right delimiters of expressions. These characters are recognized anywhere in subsequent text. For example if the left and right delimiters have both been set to "#", the input:

```
Let #x sub i#, #y# and #alpha# be positive
```

produces:

Let x_i, y and α be positive

Running a preprocessor is strikingly easy on UNIX. To typeset text stored in file "f", one issues the command:

```
eqn f | troff
```

The vertical bar connects the output of one process (EQN) to the input of another (TROFF).

5. Language Theory

The basic structure of the language is not a particularly original one. Equations are pictured as a set of "boxes," pieced together in various ways. For example, something with a subscript is just a box followed by another box moved downward and shrunk by an appropriate amount. A fraction is just a box centered above another box, at the right altitude, with a line of correct length drawn between them.

The grammar for the language is shown below. For purposes of exposition, we have collapsed some productions. In the original grammar, there are about 70 productions, but many of these are simple ones used only to guarantee that some keyword is recognized early enough in the parsing process. Symbols in capital letters are terminal symbols, lower case symbols are non-terminals, i.e., syntactic categories. The vertical bar | indicates an alternative; the brackets [ ] indicate optional material. A TEXT is a string of non-blank characters or any string inside double quotes; the other terminal symbols represent literal occurrences of the corresponding keyword.

```
eqn  : box | eqn box

box  : text
     | { eqn }
     | box OVER box
     | SQRT box
     | box SUB box | box SUP box
     | [ L | C | R ]PILE { list }
     | LEFT text eqn [ RIGHT text ]
     | box [ FROM box ] [ TO box ]
     | SIZE text box
     | [ROMAN | BOLD | ITALIC] box
     | box [HAT | BAR | DOT | DOTDOT | TILDE]
     | DEFINE text text

list : eqn | list ABOVE eqn

text : TEXT
```

The grammar makes it obvious why there are few exceptions. For example, the observation that something can be replaced by a more complicated something in braces is implicit in the productions:

```
eqn  : box | eqn box
box  : text | { eqn }
```

Anywhere a single character could be used, *any* legal construction can be used.

Clearly, our grammar is highly ambiguous. What, for instance, do we do with the input

```
a over b over c
```

?

Is it

```
{a over b} over c
```

or is it

```
a over {b over c}
```

?

To answer questions like this, the grammar is supplemented with a small set of rules that describe the precedence and associativity of operators. In particular, we specify (more or less arbitrarily) that *over* associates to the left, so the first alternative above is the one chosen. On the other hand, *sub* and *sup* bind to the right,

because this is closer to standard mathematical practice. That is, we assume x^{a^b} is $x^{(a^b)}$, not $(x^a)^b$.

The precedence rules resolve the ambiguity in a construction like

```
a sup 2 over b
```

We define *sup* to have a higher precedence than *over*, so this construction is parsed as $\frac{a^2}{b}$ instead of $a^{\frac{2}{b}}$.

Naturally, a user can always force a particular parsing by placing braces around expressions.

The ambiguous grammar approach seems to be quite useful. The grammar we use is small enough to be easily understood, for it contains none of the productions that would be normally used for resolving ambiguity. Instead the supplemental information about precedence and associativity (also small enough to be understood) provides the compiler-compiler with the information it needs to make a fast, deterministic parser for the specific language we want. When the language is supplemented by the disambiguating rules, it is in fact LR(1) and thus easy to parse[5].

The output code is generated as the input is scanned. Any time a production of the grammar is recognized, (potentially) some TROFF commands are output. For example, when the lexical analyzer reports that it has found a TEXT (i.e., a string of contiguous characters), we have recognized the production:

```
text    : TEXT
```

The translation of this is simple. We generate a local name for the string, then hand the name and the string to TROFF, and let TROFF perform the storage management. All we save is the name of the string, its height, and its baseline.

As another example, the translation associated with the production

```
box     . box OVER box
```

is:

```
Width of output box =
  slightly more than largest input width
Height of output box =
  slightly more than sum of input heights
Base of output box =
  slightly more than height of bottom input box
String describing output box =
  move down;
  move right enough to center bottom box;
  draw bottom box (i.e., copy string for bottom box
  move up; move left enough to center top box;
  draw top box (i.e., copy string for top box);
  move down and left; draw line full width;
  return to proper base line.
```

Most of the other productions have equally simple semantic actions. Picturing the output as a set of properly placed boxes makes the right sequence of positioning commands quite obvious. The main difficulty is in finding the right numbers to use for esthetically pleasing positioning.

With a grammar, it is usually clear how to extend the language. For instance, one of our users suggested a TENSOR operator, to make constructions like

$$T^{k\,l}_{m\,ij}$$

Grammatically, this is easy: it is sufficient to add a production like

```
box     : TENSOR { list }
```

Semantically, we need only juggle the boxes to the right places.

6. Experience

There are really three aspects of interest—how well EQN sets mathematics, how well it satisfies its goal of being "easy to use," and how easy it was to build.

The first question is easily addressed. This entire paper has been set by the program. Readers can judge for themselves whether it is good enough for their purposes. One of our users commented that although the output is not as good as the best hand-set material, it is still better than average, and much better than the worst. In any case, who cares? Printed books cannot compete with the birds and flowers of illuminated manuscripts on esthetic grounds, either, but they have some clear economic advantages.

Some of the deficiencies in the output could be cleaned up with more work on our part. For example, we sometimes leave too much space between a roman letter and an italic one. If we were willing to keep track of the fonts involved, we could do this better more of the

time.

Some other weaknesses are inherent in our output device. It is hard, for instance, to draw a line of an arbitrary length without getting a perceptible overstrike at one end.

As to ease of use, at the time of writing, the system has been used by two distinct groups. One user population consists of mathematicians, chemists, physicists, and computer scientists. Their typical reaction has been something like:

(1) It's easy to write, although I make the following mistakes...

(2) How do I do...?

(3) It botches the following things.... Why don't you fix them?

(4) You really need the following features...

The learning time is short. A few minutes gives the general flavor, and typing a page or two of a paper generally uncovers most of the misconceptions about how it works.

The second user group is much larger, the secretaries and mathematical typists who were the original target of the system. They tend to be enthusiastic converts. They find the language easy to learn (most are largely self-taught), and have little trouble producing the output they want. They are of course less critical of the esthetics of their output than users trained in mathematics. After a transition period, most find using a computer more interesting than a regular typewriter.

The main difficulty that users have seems to be remembering that a blank is a delimiter; even experienced users use blanks where they shouldn't and omit them when they are needed. A common instance is typing

```
f(x sub i)
```

which produces

$$f(x_i)$$

instead of

$$f(x.)$$

Since the EQN language knows no mathematics, it cannot deduce that the right parenthesis is not part of the subscript.

The language is somewhat prolix, but this doesn't seem excessive considering how much is being done, and it is certainly more compact than the corresponding TROFF commands. For example, here is the source for the continued fraction expression in Section 1 of this paper:

```
a sub 0 + b sub 1 over
  {a sub 1 + b sub 2 over
    {a sub 2 + b sub 3 over
      {a sub 3 + ... }}}
```

This is the input for the large integral of Section 1; notice the use of definitions:

```
define emx "{e sup mx}"
define mab "{m sqrt ab}"
define sa "{sqrt a}"
define sb "{sqrt b}"
int dx over {a emx - be sup -mx} "="
left { lpile {
    1 over {2 mab} "log"
        {sa emx - sb} over {sa emx + sb}
  above
    1 over mab " tanh sup -1 ( sa over sb emx )
  above
    -1 over mab " coth sup -1 ( sa over sb emx )
}
```

As to ease of construction, we have already mentioned that there are really only a few person-months invested. Much of this time has gone into two things—fine-tuning (what is the most esthetically pleasing space to use between the numerator and denominator of a fraction?), and changing things found deficient by our users (shouldn't a tilde be a delimiter?).

The program consists of a number of small, essentially unconnected modules for code generation, a simple lexical analyzer, a canned parser which we did not have to write, and some miscellany associated with input files and the macro facility. The program is now about 1600 lines of C [6], a high-level language reminiscent of BCPL. About 20 percent of these lines are "print" statements, generating the output code.

The semantic routines that generate the actual TROFF commands can be changed to accommodate other formatting languages and devices. For example, in less than 24 hours, one of us changed the entire semantic package to drive NROFF, a variant of TROFF, for typesetting mathematics on teletypewriter devices capable of reverse line motions. Since many potential users do not have access to a typesetter, but still have to type mathematics, this provides a way to get a typed version of the final output which is close enough for debugging purposes, and sometimes even for ultimate use.

7. Conclusions

We think we have shown that it is possible to do acceptably good typesetting of mathematics on a phototypesetter, with an input language that is easy to learn and use and that satisfies many users' demands. Such a package can be implemented in short order, given a compiler-compiler

and a decent typesetting program underneath.

Defining a language, and building a compiler for it with a compiler-compiler seems like the only sensible way to do business. Our experience with the use of a grammar and a compiler-compiler has been uniformly favorable. If we had written everything into code directly, we would have been locked into our original design. Furthermore, we would have never been sure where the exceptions and special cases were. But because we have a grammar, we can change our minds readily and still be reasonably sure that if a construction works in one place it will work everywhere.

Acknowledgements

We are deeply indebted to J. F. Ossanna, the author of TROFF, for his willingness to modify TROFF to make our task easier and for his continuous assistance during the development of our program. We are also grateful to A. V. Aho for help with language theory, to S. C. Johnson for aid with the compiler-compiler, and to our early users A. V. Aho, S. I. Feldman, S. C. Johnson, R. W. Hamming, and M. D. McIlroy for their constructive criticisms.

References

[1] *A Manual of Style.* 12th Edition. University of Chicago Press, 1969. p 295.

[2] *Model CAIT Phototypesetter.* Graphic Systems, Inc., Hudson, N. H.

[3] Ritchie, D. M., and Thompson, K. L., "The UNIX time-sharing system." *Comm. ACM 17,* 7 (July 1974), 365-375.

[4] Ossanna, J. F., TROFF User's Manual. Bell Laboratories Computing Science Technical Report 54, 1977.

[5] Aho, A. V., and Johnson, S. C., "LR Parsing." *Comp. Surv. 6,* 2 (June 1974), 99-124.

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

Typesetting Mathematics — User's Guide (Second Edition)

Brian W. Kernighan and Lorinda L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This is the user's guide for a system for typesetting mathematics, using the phototypesetters on the UNIX† and GCOS operating systems.

Mathematical expressions are described in a language designed to be easy to use by people who know neither mathematics nor typesetting. Enough of the language to set in-line expressions like $\lim_{x \to \pi/2} (\tan x)^{\sin 2x} = 1$ or display equations like

$$G(z) = e^{\ln G(z)} = \exp\left(\sum_{k \geq 1} \frac{S_k z^k}{k} \right) = \prod_{k \geq 1} e^{S_k z^k / k}$$

$$= \left(1 + S_1 z + \frac{S_1^2 z^2}{2!} + \cdots \right) \left(1 + \frac{S_2 z^2}{2} + \frac{S_2^2 z^4}{2^2 \cdot 2!} + \cdots \right) \cdots$$

$$= \sum_{m \geq 0} \left(\sum_{\substack{k_1, k_2, \ldots, k_m \geq 0 \\ k_1 + 2k_2 + \cdots + mk_m = m}} \frac{S_1^{k_1}}{1^{k_1} k_1!} \frac{S_2^{k_2}}{2^{k_2} k_2!} \cdots \frac{S_m^{k_m}}{m^{k_m} k_m!} \right) z^m$$

can be learned in an hour or so.

The language interfaces directly with the phototypesetting language TROFF, so mathematical expressions can be embedded in the running text of a manuscript, and the entire document produced in one process. This user's guide is an example of its output.

The same language may be used with the UNIX formatter NROFF to set mathematical expressions on DASI and GSI terminals and Model 37 teletypes.

August 15, 1978

†UNIX is a Trademark of Bell Laboratories.

Typesetting Mathematics — User's Guide (Second Edition)

Brian W. Kernighan and Lorinda L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

EQN is a program for typesetting mathematics on the Graphics Systems phototypesetters on UNIX and GCOS. The EQN language was designed to be easy to use by people who know neither mathematics nor typesetting. Thus EQN knows relatively little about mathematics. In particular, mathematical symbols like +, −, ×, parentheses, and so on have no special meanings. EQN is quite happy to set garbage (but it will look good).

EQN works as a preprocessor for the typesetter formatter, TROFF[1], so the normal mode of operation is to prepare a document with both mathematics and ordinary text interspersed, and let EQN set the mathematics while TROFF does the body of the text.

On UNIX, EQN will also produce mathematics on DASI and GSI terminals and on Model 37 teletypes. The input is identical, but you have to use the programs NEQN and NROFF instead of EQN and TROFF. Of course, some things won't look as good because terminals don't provide the variety of characters, sizes and fonts that a typesetter does, but the output is usually adequate for proofreading.

To use EQN on UNIX,

```
eqn files | troff
```

GCOS use is discussed in section 26.

2. Displayed Equations

To tell EQN where a mathematical expression begins and ends, we mark it with lines beginning .EQ and .EN. Thus if you type the lines

```
.EQ
x=y+z
.EN
```

your output will look like

$$x=y+z$$

The .EQ and .EN are copied through untouched; they are not otherwise processed by EQN. This means that you have to take care of things like centering, numbering, and so on yourself. The most common way is to use the TROFF and NROFF macro package package '−ms' developed by M. E. Lesk[3], which allows you to center, indent, left-justify and number equations.

With the '−ms' package, equations are centered by default. To left-justify an equation, use .EQ L instead of .EQ. To indent it, use .EQ I. Any of these can be followed by an arbitrary 'equation number' which will be placed at the right margin. For example, the input

```
.EQ I (3.1a)
x = f(y/2) + y/2
.EN
```

produces the output

$$x=f(y/2)+y/2 \qquad (3.1a)$$

There is also a shorthand notation so in-line expressions like π^2 can be entered without EQ and EN. We will talk about it in section 19.

3. Input spaces

Spaces and newlines within an expression are thrown away by EQN. (Normal text is left absolutely alone.) Thus between EQ and EN,

```
x=y−z
```

and

```
x = y + z
```

and

```
x
  =
    y
  + z
```

and so on all produce the same output

$$x=y+z$$

You should use spaces and newlines freely to make your input equations readable and easy to edit. In particular, very long lines are a bad idea, since they are often hard to fix if you make a mistake.

4. Output spaces

To force extra spaces into the *output*, use a tilde "~" for each space you want:

```
x~=~y~+~z
```

gives

$$x = y + z$$

You can also use a circumflex "^", which gives a space half the width of a tilde. It is mainly useful for fine-tuning. Tabs may also be used to position pieces of an expression, but the tab stops must be set by TROFF commands.

5. Symbols, Special Names, Greek

EQN knows some mathematical symbols, some mathematical names, and the Greek alphabet. For example,

```
x=2 pi int sin ( omega t)dt
```

produces

$$x=2\pi\int\sin(\omega t)\,dt$$

Here the spaces in the input are **necessary** to tell EQN that *int*, *pi*, *sin* and *omega* are separate entities that should get special treatment. The *sin*, digit 2, and parentheses are set in roman type instead of italic; *pi* and *omega* are made Greek; and *int* becomes the integral sign.

When in doubt, leave spaces around separate parts of the input. A *very* common error is to type *f(pi)* without leaving spaces on both sides of the *pi*. As a result, EQN does not recognize *pi* as a special word, and it appears as $f(pi)$ instead of $f(\pi)$.

A complete list of EQN names appears in section 23. Knowledgeable users can also use TROFF four-character names for anything EQN doesn't know about, like *\(bs* for the Bell System sign ⓑ.

6. Spaces, Again

The only way EQN can deduce that some sequence of letters might be special is if that sequence is separated from the letters on either side of it. This can be done by surrounding a special word by ordinary spaces (or tabs or newlines), as we did in the previous section.

You can also make special words stand out by surrounding them with tildes or circumflexes:

```
x~=~2~pi~int~sin~(~omega~t~)~dt
```

is much the same as the last example, except that the tildes not only separate the magic words like *sin*, *omega*, and so on, but also add extra spaces, one space per tilde:

$$x = 2\,\pi\int\sin\,(\,\omega\,t\,)\,dt$$

Special words can also be separated by braces { } and double quotes "...", which have special meanings that we will see soon.

7. Subscripts and Superscripts

Subscripts and superscripts are obtained with the words *sub* and *sup*.

```
x sup 2 + y sub k
```

gives

$$x^2+y_k$$

EQN takes care of all the size changes and vertical motions needed to make the output look right. The words *sub* and *sup* must be surrounded by spaces; *x sub2* will give you *xsub2* instead of x_2. Furthermore, don't forget to leave a space (or a tilde, etc.) to mark the end of a subscript or superscript. A common error is to say something like

```
y = (x sup 2)+1
```

which causes

$$y=(x^{2)+1}$$

instead of the intended

$$y=(x^2)+1$$

Subscripted subscripts and superscripted superscripts also work:

```
x sub i sub 1
```

is

$$x_{i_1}$$

A subscript and superscript on the same thing are printed one above the other if the subscript comes *first:*

```
x sub i sup 2
```

is

$$x_i^2$$

Other than this special case, *sub* and *sup* group to the right, so *x sup y sub z* means x^{y_z}, not $x^{y}{}_{z}$.

8. Braces for Grouping

Normally, the end of a subscript or superscript is marked simply by a blank (or tab or tilde, etc.) What if the subscript or superscript is something that has to be typed with blanks in it? In that case, you can use the braces { and } to mark the beginning and end of the subscript or superscript:

```
e sup {i omega t}
```

is

$$e^{i\omega t}$$

Rule: Braces can *always* be used to force EQN to treat something as a unit, or just to make your intent perfectly clear. Thus:

```
x sub {i sub 1} sup 2
```

is

$$x_{i_1}^2$$

with braces, but

```
x sub i sub 1 sup 2
```

is

$$x_{i_1}^{2}$$

which is rather different.

Braces can occur within braces if necessary:

```
e sup {i pi sup {rho -1}}
```

is

$$e^{i\pi^{\rho-1}}$$

The general rule is that anywhere you could use some single thing like *x*, you can use an arbitrarily complicated thing if you enclose it in braces. EQN will look after all the details of positioning it and making it the right size.

In all cases, make sure you have the right number of braces. Leaving one out or adding an extra will cause EQN to complain bitterly.

Occasionally you will have to print braces. To do this, enclose them in double quotes, like "{". Quoting is discussed in more detail in section 14.

9. Fractions

To make a fraction, use the word *over:*

```
a+b over 2c =1
```

gives

$$\frac{a+b}{2c}=1$$

The line is made the right length and positioned automatically. Braces can be used to make clear what goes over what:

```
{alpha + beta} over {sin (x)}
```

is

$$\frac{\alpha+\beta}{\sin(x)}$$

What happens when there is both an *over* and a *sup* in the same expression? In such an apparently ambiguous case, EQN does the *sup* before the *over*, so

```
-b sup 2 over pi
```

is $\frac{-b^2}{\pi}$ instead of $-b^{\frac{2}{\pi}}$ The rules which decide which operation is done first in cases like this are summarized in section 23. When in doubt, however, *use braces* to make clear what goes with what.

10. Square Roots

To draw a square root, use *sqrt:*

```
sqrt a+b + 1 over sqrt {ax sup 2 -bx+c}
```

is

$$\sqrt{a-b}+\frac{1}{\sqrt{ax^2-bx-c}}$$

Warning — square roots of tall quantities look lousy, because a root-sign big enough to cover the quantity is too dark and heavy:

```
sqrt {a sup 2 over b sub 2}
```

is

$$\sqrt{\frac{a^2}{b_2}}$$

Big square roots are generally better written as something to the power ½:

$$(a^2/b_2)^{1/2}$$

which is

```
(a sup 2 /b sub 2 ) sup half
```

11. Summation, Integral, Etc.

Summations, integrals, and similar constructions are easy:

```
sum from i=0 to {i= inf} x sup i
```

produces

$$\sum_{i=0}^{i=\infty} x^i$$

Notice that we used braces to indicate where the upper part $i=\infty$ begins and ends. No braces were necessary for the lower part $i=0$, because it contained no blanks. The braces will never hurt, and if the *from* and *to* parts contain any blanks, you must use braces around them.

The *from* and *to* parts are both optional, but if both are used, they have to occur in that order.

Other useful characters can replace the *sum* in our example:

```
int  prod  union  inter
```

become, respectively,

$$\int \quad \prod \quad \cup \quad \cap$$

Since the thing before the *from* can be anything, even something in braces, *from-to* can often be used in unexpected ways:

```
lim from {n -> inf} x sub n =0 .
```

is

$$\lim_{n\to\infty} x_n=0$$

12. Size and Font Changes

By default, equations are set in 10-point type (the same size as this guide), with standard mathematical conventions to determine what characters are in roman and what in italic. Although EQN makes a valiant attempt to use esthetically pleasing sizes and fonts, it is not perfect. To change sizes and fonts, use *size n* and *roman, italic, bold* and *fat*. Like *sub* and *sup*, size and font changes affect only the thing that follows them, and revert to the normal situation at the end of it. Thus

```
bold x y
```

is

$$\mathbf{x}y$$

and

```
size 14 bold x = y +
  size 14 {alpha + beta}
```

gives

$$\mathbf{x}=y+\alpha+\beta$$

As always, you can use braces if you want to affect something more complicated than a single letter. For example, you can change the size of an entire equation by

```
size 12 { ... }
```

Legal sizes which may follow *size* are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, 36. You can also change the size *by* a given amount; for example, you can say *size +2* to make the size two points bigger, or *size −3* to make it three points smaller. This has the advantage that you don't have to know what the current size is.

If you are using fonts other than roman, italic and bold, you can say *font X* where *X* is a one character TROFF name or number for the font. Since EQN is tuned for roman, italic and bold, other fonts may not give quite as good an appearance.

The *fat* operation takes the current font and widens it by overstriking: *fat grad* is ∇ and *fat {x sub i}* is $\mathbf{x}_i$.

If an entire document is to be in a non-standard size or font, it is a severe nuisance to have to write out a size and font change for each equation. Accordingly, you can set a "global" size or font which

thereafter affects all equations. At the beginning of any equation, you might say, for instance,

```
.EQ
gsize 16
gfont R
...
.EN
```

to set the size to 16 and the font to roman thereafter. In place of R, you can use any of the TROFF font names. The size after *gsize* can be a relative change with + or −.

Generally, *gsize* and *gfont* will appear at the beginning of a document but they can also appear thoughout a document: the global font and size can be changed as often as needed. For example, in a footnote‡ you will typically want the size of equations to match the size of the footnote text, which is two points smaller than the main text. Don't forget to reset the global size at the end of the footnote.

13. Diacritical Marks

To get funny marks on top of letters, there are several words:

x dot	$\dot{x}$
x dotdot	$\ddot{x}$
x hat	$\hat{x}$
x tilde	$\tilde{x}$
x vec	$\vec{x}$
x dyad	$\overleftrightarrow{x}$
x bar	$\bar{x}$
x under	$\underline{x}$

The diacritical mark is placed at the right height. The *bar* and *under* are made the right length for the entire construct, as in $\overline{x+y+z}$; other marks are centered.

14. Quoted Text

Any input entirely within quotes ("...") is not subject to any of the font changes and spacing adjustments normally done by the equation setter. This provides a way to do your own spacing and adjusting if needed:

```
italic "sin(x)" + sin (x)
```

is

$$sin(x) + sin(x)$$

Quotes are also used to get braces and other EQN keywords printed:

```
"{ size alpha }"
```

is

$$\{ size\ alpha \}$$

and

```
roman "{ size alpha }"
```

is

$$\{ size\ alpha \}$$

The construction "" is often used as a place-holder when grammatically EQN needs something, but you don't actually want anything in your output. For example, to make ^{2}He, you can't just type *sup 2 roman He* because a *sup* has to be a superscript *on* something. Thus you must say

```
"" sup 2 roman He
```

To get a literal quote use "\"". TROFF characters like \(*bs* can appear unquoted, but more complicated things like horizontal and vertical motions with *h* and *v* should always be quoted. (If you've never heard of *h* and *v*, ignore this section.)

15. Lining Up Equations

Sometimes it's necessary to line up a series of equations at some horizontal position, often at an equals sign. This is done with two operations called *mark* and *lineup*.

The word *mark* may appear once at any place in an equation. It remembers the horizontal position where it appeared. Successive equations can contain one occurrence of the word *lineup*. The place where *lineup* appears is made to line up with the place marked by the previous *mark* if at all possible. Thus, for example, you can say

‡Like this one, in which we have a few random expressions like x, and π. The sizes for these were set by the command *gsize* −2.

```
.EQ I
x+y mark = z
.EN
.EQ I
x lineup = 1
.EN
```

to produce

$$x+y=z$$
$$x=1$$

For reasons too complicated to talk about, when you use EQN and 'ms', use either .EQ I or .EQ L. mark and *lineup* don't work with centered equations. Also bear in mind that *mark* doesn't look ahead:

```
x mark =1
...
x+y lineup =z
```

isn't going to work, because there isn't room for the $x+y$ part after the *mark* remembers where the x is.

16. Big Brackets, Etc.

To get big brackets [ ], braces { }, parentheses ( ), and bars | | around things, use the *left* and *right* commands:

```
left { a over b + 1 right }
~=~ left ( c over d right )
+ left [ e right ]
```

is

$$\left\{\frac{a}{b}+1\right\} = \left(\frac{c}{d}\right)+\left[e\right]$$

The resulting brackets are made big enough to cover whatever they enclose. Other characters can be used besides these, but the are not likely to look very good. One exception is the *floor* and *ceiling* characters:

```
left floor x over y right floor
<= left ceiling a over b right ceiling
```

produces

$$\left\lfloor\frac{x}{y}\right\rfloor \leqslant \left\lceil\frac{a}{b}\right\rceil$$

Several warnings about brackets are in order. First, braces are typically bigger than brackets and parentheses, because they are made up of three, five, seven, etc., pieces, while brackets can be made up of two, three, etc. Second, big left and right parentheses often look poor, because the character set is poorly designed.

The *right* part may be omitted: a "left something" need not have a corresponding "right something". If the *right* part is omitted, put braces around the thing you want the left bracket to encompass. Otherwise, the resulting brackets may be too large.

If you want to omit the *left* part, things are more complicated, because technically you can't have a *right* without a corresponding *left*. Instead you have to say

```
left "" ..... right )
```

for example. The *left* "" means a "left nothing". This satisfies the rules without hurting your output.

17. Piles

There is a general facility for making vertical piles of things; it comes in several flavors. For example:

```
A ~=~ left [
  pile { a above b above c }
  ~ pile { x above y above z }
right ]
```

will make

$$A = \left[\begin{matrix} a & x \\ b & y \\ c & z \end{matrix}\right]$$

The elements of the pile (there can be as many as you want) are centered one above another, at the right height for most purposes. The keyword *above* is used to separate the pieces; braces are used around the entire list. The elements of a pile can be as complicated as needed, even containing more piles.

Three other forms of pile exist: *lpile* makes a pile with the elements left-justified; *rpile* makes a right-justified pile; and *cpile* makes a centered pile, just like *pile*. The vertical spacing between the pieces is somewhat larger for *l-*, *r-* and *cpiles* than it is for ordinary piles.

```
roman sign (x)~=~
left {
  lpile {1 above 0 above -1}
  ~ lpile
  {if~x>0 above if~x=0 above if~x<0}
```

makes

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x>0 \\ 0 & \text{if } x=0 \\ -1 & \text{if } x<0 \end{cases}$$

Notice the left brace without a matching right one.

18. Matrices

It is also possible to make matrices. For example, to make a neat array like

$$x_i \quad x^2$$
$$y_i \quad y^2$$

you have to type

```
matrix {
  ccol { x sub i above y sub i }
  ccol { x sup 2 above y sup 2 }
}
```

This produces a matrix with two centered columns. The elements of the columns are then listed just as for a pile, each element separated by the word *above*. You can also use *lcol* or *rcol* to left or right adjust columns. Each column can be separately adjusted, and there can be as many columns as you like.

The reason for using a matrix instead of two adjacent piles, by the way, is that if the elements of the piles don't all have the same height, they won't line up properly. A matrix forces them to line up, because it looks at the entire structure before deciding what spacing to use.

A word of warning about matrices — *each column must have the same number of elements in it.* The world will end if you get this wrong.

19. Shorthand for In-line Equations

In a mathematical document, it is necessary to follow mathematical conventions not just in display equations, but also in the body of the text, for example by making variable names like x italic. Although this could be done by surrounding the appropriate parts with .EQ and .EN, the continual repetition of .EQ and .EN is a nuisance. Furthermore, with '-ms', .EQ and .EN imply a displayed equation.

EQN provides a shorthand for short in-line expressions. You can define two characters to mark the left and right ends of an in-line equation, and then type expressions right in the middle of text lines. To set both the left and right characters to dollar signs, for example, add to the beginning of your document the three lines

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

Having done this, you can then say things like

> Let $alpha sub i$ be the primary variable, and let $beta$ be zero. Then we can show that $x sub 1 > = 0$.

This works as you might expect — spaces, newlines, and so on are significant in the text, but not in the equation part itself. Multiple equations can occur in a single input line.

Enough room is left before and after a line that contains in-line expressions that something like $\sum_{i=1}^{n} x_i$ does not interfere with the lines surrounding it.

To turn off the delimiters,

```
.EQ
delim off
.EN
```

Warning: don't use braces, tildes, circumflexes, or double quotes as delimiters — chaos will result.

20. Definitions

EQN provides a facility so you can give a frequently-used string of characters a name, and thereafter just type the name instead of the whole string. For example, if the sequence

```
x sub i sub 1 + y sub i sub 1
```

appears repeatedly throughout a paper, you can save re-typing it each time by defining it like this:

```
define xy 'x sub i sub 1 + y sub i sub 1'
```

This makes .xy a shorthand for whatever characters occur between the single quotes in the definition. You can use any character

instead of quote to mark the ends of the definition, so long as it doesn't appear inside the definition.

Now you can use *xy* like this:

```
.EQ
f(x) = xy ...
.EN
```

and so on. Each occurrence of *xy* will expand into what it was defined as. Be careful to leave spaces or their equivalent around the name when you actually use it, so EQN will be able to identify it as special.

There are several things to watch out for. First, although definitions can use previous definitions, as in

```
.EQ
define  xi  ' x sub i '
define  xi1  ' xi sub 1 '
.EN
```

don't define something in terms of itself A favorite error is to say

```
define  X  ' roman X '
```

This is a guaranteed disaster, since X *is* now defined in terms of itself. If you say

```
define  X  ' roman "X" '
```

however, the quotes protect the second X, and everything works fine.

EQN keywords can be redefined. You can make / mean *over* by saying

```
define  /  ' over '
```

or redefine *over* as / with

```
define  over  ' / '
```

If you need different things to print on a terminal and on the typesetter, it is sometimes worth defining a symbol differently in NEQN and EQN. This can be done with *ndefine* and *tdefine*. A definition made with *ndefine* only takes effect if you are running NEQN; if you use *tdefine*, the definition only applies for EQN. Names defined with plain *define* apply to both EQN and NEQN.

21. Local Motions

Although EQN tries to get most things at the right place on the paper, it isn't perfect, and occasionally you will need to tune the output to make it just right. Small extra horizontal spaces can be obtained with tilde and circumflex. You can also say *back n* and *fwd n* to move small amounts horizontally. *n* is how far to move in 1/100's of an em (an em is about the width of the letter 'm'.) Thus *back 50* moves back about half the width of an m. Similarly you can move things up or down with *up n* and *down n*. As with *sub* or *sup*, the local motions affect the next thing in the input, and this can be something arbitrarily complicated if it is enclosed in braces.

22. A Large Example

Here is the complete source for the three display equations in the abstract of this guide.

```
.EQ I
G(z)~mark =~ e sup { ln ~ G(z) }
~=~ exp left (
sum from k>=1 {S sub k z sup k} over k right )
~=~ prod from k>=1 e sup {S sub k z sup k /k}
.EN
.EQ I
lineup = left ( 1 + S sub 1 z +
{ S sub 1 sup 2 z sup 2 } over 2! + ... right )
left ( 1+ { S sub 2 z sup 2 } over 2
+ { S sub 2 sup 2 z sup 4 } over { 2 sup 2 cdot 2! }
+ ... right ) ...
.EN
.EQ I
lineup = sum from m>=0 left (
sum from
pile { k sub 1 ,k sub 2 ..... k sub m  >=0
above
k sub 1 +2k sub 2 + ... +mk sub m =m}
{ S sub 1 sup {k sub 1} } over {1 sup k sub 1 k sub 1 ! } ~
{ S sub 2 sup {k sub 2} } over {2 sup k sub 2 k sub 2 ! } ~
...
{ S sub m sup {k sub m} } over {m sup k sub m k sub m ! }
right ) z sup m
.EN
```

23. Keywords, Precedences, Etc.

If you don't use braces, EQN will do operations in the order shown in this list.

dyad vec under bar tilde hat dot dotdot
fwd back down up
fat roman italic bold size
sub sup sqrt over
from to

These operations group to the left:

over sqrt left right

All others group to the right.

Digits, parentheses, brackets, punctuation marks, and these mathematical words are converted to Roman font when encountered:

sin cos tan sinh cosh tanh arc
max min lim log ln exp
Re Im and if for det

These character sequences are recognized and translated as shown.

>=	≥
<=	≤
==	≡
!=	≠
+-	±
->	→
<-	←
<<	≪
>>	≫
inf	∞
partial	∂
half	½
prime	′
approx	≈
nothing	
cdot	·
times	×
del	∇
grad	∇
...	. . .
.....	. . . . .
sum	Σ
int	∫
prod	Π
union	∪
inter	∩

To obtain Greek letters, simply spell them out in whatever case you want:

DELTA	Δ	iota	ι
GAMMA	Γ	kappa	κ
LAMBDA	Λ	lambda	λ
OMEGA	Ω	mu	μ
PHI	Φ	nu	ν
PI	Π	omega	ω
PSI	Ψ	omicron	ο
SIGMA	Σ	phi	φ
THETA	Θ	pi	π
UPSILON	Υ	psi	ψ
XI	Ξ	rho	ρ
alpha	α	sigma	σ
beta	β	tau	τ
chi	χ	theta	θ
delta	δ	upsilon	υ
epsilon	ε	xi	ξ
eta	η	zeta	ζ
gamma	γ		

These are all the words known to EQN (except for characters with names), together with the section where they are discussed.

above	17, 18	lpile	17
back	21	mark	15
bar	13	matrix	18
bold	12	ndefine	20
ccol	18	over	9
col	18	pile	17
cpile	17	rcol	18
define	20	right	16
delim	19	roman	12
dot	13	rpile	17
dotdot	13	size	12
down	21	sqrt	10
dyad	13	sub	7
fat	12	sup	7
font	12	tdefine	20
from	11	tilde	13
fwd	21	to	11
gfont	12	under	13
gsize	12	up	21
hat	13	vec	13
italic	12	~ , ^	4, 6
lcol	18	{ }	8
left	16	"..."	8, 14
lineup	15		

24. Troubleshooting

If you make a mistake in an equation, like leaving out a brace (very common) or having one too many (very common) or having a *sup* with nothing before it (common), EQN will tell you with the message

syntax error between lines x and y, file z

where *x* and *y* are approximately the lines between which the trouble occurred, and *z* is the name of the file in question. The line numbers are approximate — look nearby as well. There are also self-explanatory messages that arise if you leave out a quote or try to run EQN on a non-existent file.

If you want to check a document before actually printing it (on UNIX only),

```
eqn files >/dev/null
```

will throw away the output but print the messages.

If you use something like dollar signs as delimiters, it is easy to leave one out. This causes very strange troubles. The program *checkeq* (on GCOS, use *./checkeq* instead) checks for misplaced or missing dollar signs and similar troubles.

In-line equations can only be so big because of an internal buffer in TROFF. If you get a message "word overflow", you have exceeded this limit. If you print the equation as a displayed equation this message will usually go away. The message "line overflow" indicates you have exceeded an even bigger buffer. The only cure for this is to break the equation into two separate ones.

On a related topic, EQN does not break equations by itself — you must split long equations up across multiple lines by yourself, marking each by a separate .EQ ... .EN sequence. EQN does warn about equations that are too long to fit on one line.

25. Use on UNIX

To print a document that contains mathematics on the UNIX typesetter,

```
eqn files | troff
```

If there are any TROFF options, they go after the TROFF part of the command. For example,

```
eqn files | troff -ms
```

To run the same document on the GCOS typesetter, use

```
eqn files | troff -g (other options) | gcat
```

A compatible version of EQN can be used on devices like teletypes and DASI and GSI terminals which have half-line forward and reverse capabilities. To print equations on a Model 37 teletype, for example, use

```
neqn files | nroff
```

The language for equations recognized by NEQN is identical to that of EQN, although of course the output is more restricted.

To use a GSI or DASI terminal as the output device,

```
neqn files | nroff -T.x
```

where *.x* is the terminal type you are using, such as *300* or *300S*.

EQN and NEQN can be used with the TBL program [2] for setting tables that contain mathematics. Use TBL before [N]EQN, like this:

```
tbl files | eqn | troff
tbl files | neqn | nroff
```

26. Acknowledgments

We are deeply indebted to J. F. Ossanna, the author of TROFF, for his willingness to extend TROFF to make our task easier, and for his continuous assistance during the development and evolution of EQN. We are also grateful to A. V. Aho for advice on language design, to S. C. Johnson for assistance with the YACC compiler-compiler, and to all the EQN users who have made helpful suggestions and criticisms.

References

[1] J. F. Ossanna, "NROFF/TROFF User's Manual", Bell Laboratories Computing Science Technical Report #54, 1976.

[2] M. E. Lesk, "Typing Documents on UNIX", Bell Laboratories, 1976.

[3] M. E. Lesk, "TBL — A Program for Setting Tables", Bell Laboratories Computing Science Technical Report #49, 1976.

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 begining 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 .hl 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
.hl
Text of keep to be floated.
.sp
.ce
Figure 1. Example of a Floating Keep.
.hl
.)z
```

Figure 1. Example of a Floating Keep.

```
first line of unfilled display
more lines
```

Typing the character L after the .(l request produces the left justified result:

first line of unfilled display
more lines

Using C instead of L produces the line-at-a-time centered output:

first line of unfilled display
more lines

Sometimes it may be that you want to center several lines as a group, rather than centering them one line at a time. To do this use centered blocks, which are surrounded by the requests .(c and .)c. All the lines are centered as a unit, such that the longest line is centered and the rest are lined up around that line. Notice that lines do not move relative to each other using centered blocks, whereas they do using the C argument to keeps.

Centered blocks are *not* keeps, and may be used in conjunction with keeps. For example, to center a group of lines as a unit and keep them on one page, use:

```
.(b L
.(c
first line of unfilled display
more lines
.)c
.)b
```

to produce:

first line of unfilled display
more lines

If the block requests (.(b and .)b) had been omitted the result would have been the same, but with no guarantee that the lines of the centered block would have all been on one page. Note the use of the L argument to .(b; this causes the centered block to center within the entire line rather than within the line minus the indent. Also, the center requests must be nested *inside* the keep requests.

4. Annotations

There are a number of requests to save text for later printing. *Footnotes* are printed at the bottom of the current page. *Delayed text* is intended to be a variant form of footnote: the text is printed only when explicitly called for, such as at the end of each chapter. *Indexes* are a type of delayed text having a tag (usually the page number) attached to each entry after a row of dots. Indexes are also saved until called for explicitly.

4.1. Footnotes

Footnotes begin with the request .(f and end with the request .)f. The current footnote number is maintained automatically, and can be used by typing **, to produce a footnote number[1]. The number is automatically incremented after every footnote. For example, the input:

[1] Like this

```
.(q
A man who is not upright
and at the same time is presumptuous;
one who is not diligent and at the same time is ignorant;
one who is untruthful and at the same time is incompetent;
such men I do not count among acquaintances.\**
.(f
\**James R. Ware,
.ul
The Best of Confucius,
Halcyon House, 1950.
Page 77.
.)f
.)q
```

generates the result:

> A man who is not upright and at the same time is presumptuous; one who is not diligent and at the same time is ignorant; one who is untruthful and at the same time is incompetent; such men I do not count among acquaintances.[2]

It is important that the footnote appears *inside* the quote, so that you can be sure that the footnote will appear on the same page as the quote.

4.2. Delayed Text

Delayed text is very similar to a footnote except that it is printed when called for explicitly. This allows a list of references to appear (for example) at the end of each chapter, as is the convention in some disciplines. Use *# on delayed text instead of ** as on footnotes.

If you are using delayed text as your standard reference mechanism, you can still use footnotes, except that you may want to reference them with special characters* rather than numbers.

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:

---

[2] 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:

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 "l" 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.

The -me macros described in this document are not supported by Computing Services, but the package is used by some experienced UNIX users as an alternative to our supported product, the -ms macro package (document UNX 4.3.2).

−ME REFERENCE MANUAL

Release 1.1l20

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 **−rx1** 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 session) *initializes* the macro processor. After initialization it is not possible to use any of the following requests: **.sc**, **.lo**, **.th**, or **.ac**. 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 similiar 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** [8] 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 preceeded 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**.

.(l *m f*	Begin list. Lists are single spaced, unfilled text. If *f* is **F**, the list will be filled. If *m* [I] 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], preceeded 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 preceeded 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**).

.r *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**.

.n1 — 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 I, and left justified if *m* is L. *T* is a title printed on the right margin next to the equation. See *Typesetting Mathematics — User's Guide* by Brian W. Kernighan and Lorinda L. Cherry.

.EN *c* — End equation. If *c* is C the equation must be continued by immediately following with another .EQ, the text of which can be centered along with this one. Otherwise, the equation is printed, always on one page, with \n(es [0.5v in TROFF, 1v in NROFF] space above and below it.

.TS *h*
Table start. Tables are single spaced and kept on one page if possible. If you have a large table which will not fit on one page, use *h* = **H** and follow the header part (to be printed on every page of the table) with a **.TH**. See *Tbl − A Program to Format Tables* by M. E. Lesk.

.TH
With **.TS H**, ends the header portion of the table.

.TE
Table end. Note that this table does not float, in fact, it is not even guaranteed to stay on one page if you use requests such as **.sp** intermixed with the text of the table. If you want it to float (or if you use requests inside the table), surround the entire table (including the **.TS** and **.TE** requests) with the requests **.(z** and **.)z**.

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**.

.xl +*N*
Set the line length to *N* [6.0i]. This differs from **.ll** because it only affects the current environment.

.ll +*N*
Set line length in all environments to *N* [6.0i]. This should not be used after output has begun, and particularly not in two-columned output. The current line length is stored in **\n(Sl**.

.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 preceeded 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(Sf*]. 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.

*(td — 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.
*−	¾ em dash in TROFF; two hyphens in NROFF.

13. Special Characters and Marks

There are a number of special characters and diacritical marks (such as accents) available through −me. To reference these characters, you must call the macro .sc to define the characters before using them.

.sc	Define special characters and diacritical marks, as described in the remainder of this section. This macro must be stated before initialization.

The special characters available are listed below.

Name	Usage	Example	
Acute accent	*'	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.

PWB/MM
Programmer's Workbench
Memorandum Macros

D. W. Smith
J. R. Mashey
E. C. Pariser (January 1980 Reissue)

Bell Telephone Laboratories, Incorporated

PWB/MM
Programmer's Workbench Memorandum Macros

LIST OF FIGURES

PWB/MM—Programmer's Workbench Memorandum Macros

D. W. Smith
J. R. Mashey
E. C. Pariser (January 1980 Reissue)

Bell Laboratories
Piscataway, New Jersey 08854

1. INTRODUCTION

1.1 Purpose

This memorandum is the user's guide and reference manual for PWB/MM (or just **-mm**), a general-purpose package of text formatting macros for use with the UNIX† text formatters *nroff* [9] and *troff* [9]. The purpose of PWB/MM is to provide to the users of PWB/UNIX a unified, consistent, and flexible tool for producing many common types of documents. Although PWB/UNIX provides other macro packages for various *specialized* formats, PWB/MM has become the standard, general-purpose macro package for most documents.

PWB/MM can be used to produce:

- Letters.
- Reports.
- Technical Memoranda.
- Released Papers.
- Manuals.
- Books.
- etc.

The uses of PWB/MM range from single-page letters to documents of several hundred pages in length, such as user guides, design proposals, etc.

1.2 Conventions

Each section of this memorandum explains a single facility of PWB/MM. In general, the earlier a section occurs, the more necessary it is for most users. Some of the later sections can be completely ignored if PWB/MM defaults are acceptable. Likewise, each section progresses from normal-case to special-case facilities. We recommend reading a section in detail only until there is enough information to obtain the desired format, then skimming the rest of it, because some details may be of use to just a few people.

Numbers enclosed in curly brackets ({}) refer to section numbers within this document. For example, this is {1.2}.

Sections that require knowledge of the formatters {1.4} have a bullet (•) at the end of the section heading.

In the synopses of macro calls, square brackets ([]) surrounding an argument indicate that it is optional. Ellipses (...) show that the preceding argument may appear more than once.

A reference of the form *name*(N) points to page *name* in section *N* of the *PWB/UNIX User's Manual* [1].

---

† UNIX is a Trademark of Bell Laboratories.

The examples of *output* in this manual are as produced by *troff*; *nroff* output would, of course, look somewhat different (Appendix D shows *both* the *nroff* and *troff* output for a simple letter). In those cases in which the behavior of the two formatters is truly different, the *nroff* action is described first, with the *troff* action following in parentheses. For example:

The title is underlined (bold).

means that the title is underlined in *nroff* and bold in *troff*.

1.3 Overall Structure of a Document

The input for a document that is to be formatted with PWB/MM possesses four major segments, any of which may be omitted; if present, they *must* occur in the following order:

- *Parameter-setting*—This segment sets the general style and appearance of a document. The user can control page width, margin justification, numbering styles for headings and lists, page headers and footers [9], and many other properties of the document. Also, the user can add macros or redefine existing ones. This segment can be omitted entirely if one is satisfied with default values; it produces no actual output, but only performs the setup for the rest of the document.
- *Beginning*—This segment includes those items that occur only once, at the beginning of a document, e.g., title, author's name, date.
- *Body*—This segment is the actual text of the document. It may be as small as a single paragraph, or as large as hundreds of pages. It may have a hierarchy of *headings* up to seven levels deep [4]. Headings are automatically numbered (if desired) and can be saved to generate the table of contents. Five additional levels of subordination are provided by a set of *list* macros for automatic numbering, alphabetic sequencing, and "marking" of list items [5]. The body may also contain various types of displays, tables, figures, and footnotes [7, 8].
- *Ending*—This segment contains those items that occur once only, at the end of a document. Included here are signature(s) and lists of notations (e.g., "copy to" lists) [6.12]. Certain macros may be invoked here to print information that is wholly or partially derived from the rest of the document, such as the table of contents or the cover sheet for a document [10].

The existence and size of these four segments varies widely among different document types. Although a specific item (such as date, title, author name(s), etc.) may be printed in several different ways depending on the document type, there is a uniform way of typing it in.

1.4 Definitions

The term *formatter* refers to either of the text-formatting programs *nroff* and *troff*.

Requests are built-in commands recognized by the formatters. Although one seldom needs to use these requests directly [3.9], this document contains references to some of them. Full details are given in [9]. For example, the request:

```
.sp
```

inserts a blank line in the output.

Macros are named collections of requests. Each macro is an abbreviation for a collection of requests that would otherwise require repetition. PWB/MM supplies many macros, and the user can define additional ones. Macros and requests share the same set of names and are used in the same way.

Strings provide character variables, each of which names a string of characters. Strings are often used in page headers, page footers, and lists. They share the pool of names used by *requests* and *macros*. A string can be given a value via the .ds (define string) request, and its value can be obtained by referencing its name, preceded by "*" (for 1-character names) or "*(" (for 2-character names). For instance, the string *DT* in PWB/MM normally contains the current date, so that the *input* line:

```
Today is \*(DT.
```

may result in the following *output:*

```
Today is January 22, 1980.
```

The current date can be replaced, e.g.:

```
.ds DT 01/01/79
```

or by invoking a macro designed for that purpose {6.7.1}.

Number registers fill the role of integer variables. They are used for flags, for arithmetic, and for automatic numbering. A register can be given a value using a .nr request, and be referenced by preceding its name by "\n" (for 1-character names) or "\n(" (for 2-character names). For example, the following sets the value of the register *d* to 1 more than that of the register *dd*:

```
.nr d 1+\n(dd
```

See {13.1} regarding naming conventions for requests, macros, strings, and number registers.

1.5 Prerequisites and Further Reading

1.5.1 Prerequisites. We assume familiarity with UNIX at the level given in [3] and [4]. Some familiarity with the request summary in [9] is helpful.

1.5.2 Further Reading. [9] provides detailed descriptions of formatter capabilities, while [5] provides a general overview. See [6] (and possibly [7]) for instructions on formatting mathematical expressions. See *tbl*(1) and [11] for instructions on formatting tabular data.

Examples of formatted documents and of their respective input, as well as a quick reference to the material in this manual are given in [8].

2. INVOKING THE MACROS

This section tells how to access PWB/MM, shows PWB/UNIX command lines appropriate for various output devices, and describes command-line flags for PWB/MM. Note that file names, program names, and typical command sequences apply only to PWB/UNIX; different names and command lines may have to be used on other systems.

2.1 The mm Command

The *mm*(1) command can be used to print documents using *nroff* and PWB/MM; this command invokes *nroff* with the -mm flag {2.2}. It has options to specify preprocessing by *tbl*(1) and/or by *neqn*(1), and for postprocessing by various output filters. Any arguments or flags that are not recognized by *mm*(1), e.g. -rC3, are passed to *nroff* or to PWB/MM, as appropriate. The options, which can occur in any order but *must* appear before the file names, are:

-e	*neqn*(1) is to be invoked.
-t	*tbl*(1) is to be invoked.
-c	*col*(1) is to be invoked.
-E	the "-e" option of *nroff* is to be invoked.
-12	need 12-pitch mode. Be sure that the pitch switch on the terminal is set to 12.
-T300	output is to a DASI300 terminal. This is the *default* terminal type (unless STERM is set).
-T300-12	output is to a DASI300 in 12-pitch mode.
-T300s	output is to a DASI300S.
-T300S	output is to a DASI300S.
-T300s-12	output is to a DASI300S in 12-pitch mode.
-T300S-12	output is to a DASI300S in 12-pitch mode.
-T4014	output is to a Tektronix 4014.
-Thp	output is to a HP264x.

-T450 output is to a DASI450.
-T450-12 output is to a DASI450 in 12-pitch mode.
-Tln output is to a GE TermiNet 300.
-Tln300 output is to a GE TermiNet 300.
-Tti output is to a Texas Instrument 700 series terminal.
-T37 output is to a TELETYPE® Model 37.
-T43 output is to a TELETYPE® Model 43.

2.2 The -mm Flag

The PWB/MM package can also be invoked by including the **-mm** flag as an argument to the formatter. It causes the file /usr/lib/tmac/tmac.m to be read and processed before any other files. This action defines the PWB/MM macros, sets default values for various parameters, and initializes the formatter to be ready to process the files of input text.

2.3 Typical Command Lines

The prototype command lines are as follows (with the various options explained in [2.4] and in [9]).

- Text without tables or equations:

```
mm [options] filename ...
or nroff [options] -mm filename ...
or troff [options] -mm filename ...
```

- Text with tables:

```
mm -t [options] filename ...
or tbl filename ... | nroff [options] -mm
or tbl filename ... | troff [options] -mm
```

- Text with equations:

```
mm -e [options] filename ...
or neqn filename ... | nroff [options] -mm
or eqn filename ... | troff [options] -mm
```

- Text with both tables and equations:

```
mm -t -e [options] filename ...
or tbl filename ... | neqn | nroff [options] -mm
or tbl filename ... | eqn | troff [options] -mm
```

When formatting a document with *nroff*, the output should normally be processed for a specific type of terminal, because the output may require some features that are specific to a given terminal, e.g., reverse paper motion or half-line paper motion in both directions. Some commonly-used terminal types and the command lines appropriate for them are given below. See [2.4] as well as *300*(1), *450*(1), *hp*(1), *col*(1), and *terminals*(7) for further information.

- DASI300 (GSI300/DTC300) in 10-pitch, 6 lines/inch mode and a line length of 65 characters:

```
mm filename ...
or nroff -T300 -h -mm filename ...
```

- DASI300 (GSI300/DTC300) in 12-pitch, 6 lines/inch mode and a line length of 80—rather than 65—characters:

```
mm -12 filename ...
or nroff -T300-12 -rW80 -rO3 -h -mm filename ...
```

or, equivalently (and more succinctly):

nroff -T300-12 -rT1 -h -mm filename ...

- DASI450 in 10-pitch, 6 lines/inch mode:

  mm -T450 filename ...
  or **nroff -T450 -h -mm** filename ...

- DASI450 in 12-pitch, 6 lines/inch mode:

  mm -T450-12 filename ...
  or **nroff -T450-12 -rW80 -rO3 -h -mm** filename ...
  or **nroff -T450-12 -rT1 -h -mm** filename ...

- Hewlett-Packard HP264x CRT family:

  mm -Thp filename ...
  or **nroff -h -mm** filename ... **| hp**

- Any terminal incapable of reverse paper motion (GE TermiNet, Texas Instruments 700 series, etc.):

  mm -Ttn filename ...
  or **nroff -mm** filename ... **| col**

- Versatec printer (see *vp*(1) for additional details):

  vp [vp-options] "**mm -rT2 -c** filename ..."
  or **vp** [vp-options] "**nroff -rT2 -mm** filename ... **| col**"

Of course, *tbl*(1) and *eqn*(1)/*neqn*(1), if needed, must be invoked as shown in the command line prototypes at the beginning of this section.

If two-column processing {11.4} is used with *nroff*, either the -c option must be specified to *mm*(1), or the *nroff* output must be postprocessed by *col*(1). In the latter case, the -T37 terminal type must be specified to *nroff*, the -h option must *not* be specified, and the output of *col*(1) must be processed by the appropriate terminal filter (e.g., *300*(1)); *mm*(1) with the -c option handles all this automatically.

2.4 Parameters that Can Be Set from the Command Line

Number registers are commonly used within PWB/MM to hold parameter values that control various aspects of output style. Many of these can be changed within the text files via .nr requests. In addition, some of these registers can be set from the command line itself, a useful feature for those parameters that should *not* be permanently embedded within the input text itself. If used, these registers (with the possible exception of the register *P*—see below) *must* be set on the command line (or before the PWB/MM macro definitions are processed) and their meanings are:

-rA1 has the effect of invoking the .AF macro without an argument {6.7.2}.

-rB*n* defines the macros for the cover sheet and the table of contents. If *n* is 1, table-of-contents processing is enabled. If *n* is 2, then cover-sheet processing will occur. If *n* is 3, both will occur. That is, *B* having a value greater than 0 *defines* the .TC {10.1} and/or .CS {10.2} macros. Note that to have any effect, these macros must also be *invoked*.

-rC*n* *n* sets the type of copy (e.g., DRAFT) to be printed at the bottom of each page. See {9.5}.
n = 1 for OFFICIAL FILE COPY.
n = 2 for DATE FILE COPY.
n = 3 for DRAFT.

-rD1 sets *debug mode*. This flag requests the formatter to attempt to continue processing even if PWB/MM detects errors that would otherwise cause termination. It also includes some debugging information in the default page header {9.2, 11.3}.

-rL*k* sets the length of the physical page to *k* lines.[1] The default value is 66 lines per page. This parameter is used for obtaining 8 lines-per-inch output on 12-pitch terminals, or when directing output to a Versatec printer.

-rN*n* specifies the page numbering style. When *n* is 0 (default), all pages get the (prevailing) header {9.2}. When *n* is 1, the page header replaces the footer on page 1 only. When *n* is 2, the page header is omitted from page 1. When *n* is 3, "section-page" numbering {4.5} occurs. When *n* is 4, the *default* page header is suppressed; however a user-specified header is not affected.

n	*Page 1*	*Pages 2 ff.*
0	header	header
1	header replaces footer	header
2	no header	header
3	"section-page" as *footer*	
4	no header	no header unless PH defined

The contents of the prevailing header and footer do *not* depend on of the value of the number register *N*; *N* only controls whether and where the header (and, for *N*= 3, the footer) is printed, as well as the page numbering style. In particular, if the header and footer are null {9.2, 9.5}, the value of *N* is irrelevant.

-rO*k* offsets output *k* spaces to the right.[1] It is helpful for adjusting output positioning on some terminals. NOTE: The register name is the capital letter "O", *not* the digit zero (0).

-rP*n* specifies that the pages of the document are to be numbered starting with *n*. This register may also be set via a .nr request in the input text.

-rS*n* sets the point size and vertical spacing for the document. The default *n* is 10, i.e., 10-point type on 12-point leading (vertical spacing), giving 6 lines per inch {11.8}. This parameter applies to *troff* only.

-rT*n* provides register settings for certain devices. If *n* is 1, then the line length and page offset are set for output directed to a DASI300 or DASI450 in 12-pitch, 6 lines/inch mode, i.e., they are set to 80 and 3, respectively. Setting *n* to 2 changes the page length to 84 lines per page and inhibits underlining; it is meant for output sent to the Versatec printer. The default value for *n* is 0. This parameter applies to *nroff* only.

-rU1 controls underlining of section headings. This flag causes only letters and digits to be underlined. Otherwise, all characters (including spaces) are underlined {4.2.2.4.2}. This parameter applies to *nroff* only.

-rW*k* page width (i.e., line length and title length) is set to *k*.[1] This can be used to change the page width from the default value of 65 characters (6.5 inches).

2.5 Omission of -mm

If a large number of arguments is required on the command line, it may be convenient to set up the first (or only) input file of a document as follows:

```
zero or more initializations of registers listed in {2.4}
.so /usr/lib/tmac/tmac.m
remainder of text
```

1. For *nroff*, *k* is an *unscaled* number representing lines or character positions; for *troff*, *k* must be *scaled*.

In this case, one must *not* use the -mm flag (nor the *mm*(1) command); the .so request has the equivalent effect, but the registers in {2.4} must be initialized *before* the .so request, because their values are meaningful only if set before the macro definitions are processed. When using this method, it is best to "lock" into the input file only those parameters that are seldom changed. For example:

```
.nr W 80
.nr O 10
.nr N 3
.nr B 1
.so /usr/lib/tmac/tmac.m
.H 1 "INTRODUCTION"
   :
   :
```

specifies, for *nroff*, a line length of 80, a page offset of 10, "section-page" numbering, and table of contents processing.

3. FORMATTING CONCEPTS

3.1 Basic Terms

The normal action of the formatters is to *fill* output lines from one or more input lines. The output lines may be *justified* so that both the left and right margins are aligned. As the lines are being filled, words are hyphenated {3.4} as necessary. It is possible to turn any of these modes on and off (see .SA {11.2}, *Hy* {3.4}, and the formatter .nf and .fi requests {9}). Turning off fill mode also turns off justification and hyphenation.

Certain formatting commands (requests and macros) cause the filling of the current output line to cease, the line (of whatever length) to be printed, and the subsequent text to begin a new output line. This printing of a partially filled output line is known as a *break*. A few formatter requests and most of the PWB/MM macros cause a break.

While formatter requests can be used with PWB/MM, one must fully understand the consequences and side-effects that each such request might have. Actually, there is little need to use formatter requests; the macros described here should be used in most cases because:

- it is much easier to control (and change at any later point in time) the overall style of the document.
- complicated facilities (such as footnotes or tables of contents) can be obtained with ease.
- the user is insulated from the peculiarities of the formatter language.

A good rule is to use formatter requests only when absolutely necessary {3.9}.

In order to make it easy to revise the input text at a later time, input lines should be kept short and should be broken at the end of clauses; each new full *sentence must* begin on a new line.

3.2 Arguments and Double Quotes

For any macro call, a *null argument* is an argument whose width is zero. Such an argument often has a special meaning; the preferred form for a null argument is "". Note that *omitting* an argument is *not* the same as supplying a *null argument* (for example, see the .MT macro in {6.6}). Furthermore, omitted arguments can occur only at the end of an argument list, while null arguments can occur anywhere.

Any macro argument containing ordinary (paddable) spaces *must* be enclosed in double quotes (").[2] Otherwise, it will be treated as several separate arguments.

---

2. A double quote (") is a *single* character that must not be confused with two apostrophes or acute accents ('') or with two grave accents (``).

Double quotes (") are *not* permitted as part of the value of a macro argument or of a string that is to be used as a macro argument. If you must, use two grave accents (``) and/or two acute accents ('') instead. This restriction is necessary because many macro arguments are processed (interpreted) a variable number of times; for example, headings are first printed in the text and may be (re)printed in the table of contents.

3.3 Unpaddable Spaces

When output lines are *justified* to give an even right margin, existing spaces in a line may have additional spaces appended to them. This may harm the desired alignment of text. To avoid this problem, it is necessary to be able to specify a space that cannot be expanded during justification, i.e., an *unpaddable space*. There are several ways to accomplish this.

First, one may type a backslash followed by a space ("\ "). This pair of characters directly generates an *unpaddable space*. Second, one may sacrifice some seldom-used character to be translated into a space upon output. Because this translation occurs after justification, the chosen character may be used anywhere an unpaddable space is desired. The tilde (~) is often used for this purpose. To use it in this way, insert the following at the beginning of the document:

```
.tr ~
```

If a tilde must actually appear in the output, it can be temporarily "recovered" by inserting:

```
.tr ~~
```

before the place where it is needed. Its previous usage is restored by repeating the ".tr ~~", but only after a break or after the line containing the tilde has been forced out. Note that the use of the tilde in this fashion is *not* recommended for documents in which the tilde is used within equations.

3.4 Hyphenation

The formatters (and, therefore, PWB/MM) will automatically hyphenate words, if need be. However, the user may specify the hyphenation points for a specific occurrence of any word by the use of a special character known as a hyphenation indicator, or may specify hyphenation points for a small list of words (about 128 characters).

If the *hyphenation indicator* (initially, the two-character sequence "\%") appears at the beginning of a word, the word is *not* hyphenated. Alternatively, it can be used to indicate legal hyphenation point(s) inside a word. In any case, *all* occurrences of the hyphenation indicator disappear on output.

The user may specify a different hyphenation indicator:

```
.HC [hyphenation-indicator]
```

The circumflex (^) is often used for this purpose; this is done by inserting the following at the beginning of a document:

```
.HC ^
```

Note that any word containing hyphens or dashes—also known as *em* dashes—will be hyphenated immediately after a hyphen or dash if it is necessary to hyphenate the word, *even if the formatter hyphenation function is turned off.*

Hyphenation can be turned off in the body of the text by specifying:

```
.nr Hy 0
```

once at the beginning of the document. For hyphenation control within footnote text and across pages, see {8.3}.

The user may supply, via the .hw request, a small list of words with the proper hyphenation points indicated. For example, to indicate the proper hyphenation of the word "printout," one may specify:

.hw print-out

3.5 Tabs

The macros .MT {6.6}, .TC {10.1}, and .CS {10.2} use the formatter .ta request to set tab stops, and then restore the *default* values[3] of tab settings. Thus, setting tabs to other than the default values is the user's responsibility.

Note that a tab character is always interpreted with respect to its position on the *input line*, rather than its position on the output line. In general, tab characters should appear only on lines processed in "no-fill" mode {3.1}.

Also note that *tbl*(1) {7.3} changes tab stops, but does *not* restore the default tab settings.

3.6 Special Use of the BEL Character

The non-printing character BEL is used as a delimiter in many macros where it is necessary to compute the width of an argument or to delimit arbitrary text, e.g., in headers and footers {9}, headings {4}, and list marks {5}. Users who include BEL characters in their input text (especially in arguments to macros) will receive mangled output.

3.7 Bullets

A bullet (•) is often obtained on a typewriter terminal by using an "o" overstruck by a "+". For compatibility with *troff*, a bullet string is provided by PWB/MM. Rather than overstriking, use the sequence:

*(BU

wherever a bullet is desired. Note that the bullet list (.BL) macros {5.3.3.2} use this string to automatically generate the bullets for the list items.

3.8 Dashes, Minus Signs, and Hyphens

Troff has distinct graphics for a dash, a minus sign, and a hyphen, while *nroff* does not. Those who intend to use *nroff* only may use the minus sign ("-") for all three.

Those who wish mainly to use *troff* should follow the escape conventions of {9}.

Those who want to use both formatters must take care during text preparation. Unfortunately, these characters cannot be represented in a way that is both compatible and convenient. We suggest the following approach:

Dash — Type *(EM for each text dash for both *nroff* and *troff*. This string generates an em dash (—) in *troff* and generates "--" in *nroff*. Note that the dash list (.DL) macros {5.3.3.3} automatically generate the em dashes for the list items.

Hyphen — Type "-" and use as is for both formatters. *Nroff* will print it as is, and *troff* will print a true hyphen.

Minus — Type "\-" for a true minus sign, regardless of formatter. *Nroff* will effectively ignore the "\", while *troff* will print a true minus sign.

3.9 Trademark String

A trademark string *(Tm is now available with PWB/MM. This places the letters "TM" one-half line above the text that it follows.

---

3. Every eight characters in *nroff*; every ½ inch in *troff*.

For example:

```
The PWB/UNIX\*(Tm User's Manual is available from the library.
```

yields:
The PWB/UNIX™ User's Manual is available from the library.

3.10 Use of Formatter Requests

Most formatter requests [9] should *not* be used with PWB/MM because PWB/MM provides the corresponding formatting functions in a much more user-oriented and surprise-free fashion than do the basic formatter requests (3.1). However, some formatter requests *are* useful with PWB/MM, namely:

```
.af     .br     .ce     .de     .ds     .fi     .hw     .ls     .nf     .nr
.nx     .rm     .rr     .rs     .so     .sp     .ta     .ti     .tl     .tr
```

The .fp, .lg, and .ss requests are also sometimes useful for *troff*. Use of other requests without fully understanding their implications very often leads to disaster.

4. PARAGRAPHS AND HEADINGS

This section describes simple paragraphs and section headings. Additional paragraph and list styles are covered in [5].

4.1 Paragraphs

```
.P [type]
one or more lines of text.
```

This macro is used to begin two kinds of paragraphs. In a *left-justified* paragraph, the first line begins at the left margin, while in an *indented* paragraph, it is indented five spaces (see below).

A document possesses a *default paragraph style* obtained by specifying ".P" before each paragraph that does *not* follow a heading (4.2). The default style is controlled by the register *Pt*. The initial value of *Pt* is 0, which always provides left-justified paragraphs. All paragraphs can be forced to be indented by inserting the following at the beginning of the document:

```
.nr Pt 1
```

All paragraphs will be indented except after headings, lists, and displays if the following:

```
.nr Pt 2
```

is inserted at the beginning of the document.

The amount a paragraph is indented is contained in the register *Pi*, whose default value is 5. To indent paragraphs by, say, 10 spaces, insert:

```
.nr Pi 10
```

at the beginning of the document. Of course, both the *Pi* and *Pt* register values must be greater than zero for any paragraphs to be indented.

The number register *Ps* controls the amount of spacing between paragraphs. By default, *Ps* is set to 1, yielding one blank space (½ a vertical space).

☛ *Values that specify indentation must be* unscaled *and are treated as "character positions," i.e., as a number of* ens. *In* troff, *an* en *is the number of points (1 point = 1/72 of an inch) equal to half the current point size. In* nroff, *an* en *is equal to the width of a character.*

Regardless of the value of *Pt*, an *individual* paragraph can be forced to be left-justified or indented. ".P 0" always forces left justification; ".P 1" always causes indentation by the amount specified by the register *Pi*.

If .P occurs inside a *list*, the indent (if any) of the paragraph is added to the current list indent {5}.

4.2 Numbered Headings

.H level [heading-text]
zero or more lines of text

The .H macro provides seven levels of numbered headings, as illustrated by this document. Level 1 is the most major or highest; level 7 the lowest.

☛ *There is no need for a .P macro after a .H (or .HU {4.3}), because the .H macro also performs the function of the .P macro. In fact, if a .P follows a .H, the .P is ignored.* {4.2.2.2}.

4.2.1 Normal Appearance. The normal appearance of headings is as shown in this document. The effect of .H varies according to the *level* argument. First-level headings are *preceded* by two blank lines (one vertical space); all others are *preceded* by one blank line (½ a vertical space).

.H 1 heading-text — gives an underlined (bold) heading *followed* by a single blank line (½ a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Full capital letters should normally be used to make the heading stand out.

.H 2 heading-text — yields an underlined (bold) heading followed by a single blank line (½ a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Normally, initial capitals are used.

.H *n* heading-text — for $3 \leqslant n \leqslant 7$, produces an underlined (italic) heading followed by two spaces. The following text appears on the same line, i.e., these are *run-in* headings.

Appropriate numbering and spacing (horizontal and vertical) occur even if the heading text is omitted from a .H macro call.

Here are the first few .H calls of {4}:

```
.H 1 "PARAGRAPHS AND HEADINGS"
.H 2 "Paragraphs"
.H 2 "Numbered Headings"
.H 3 "Normal Appearance."
.H 3 "Altering Appearance of Headings."
.H 4 "Pre-Spacing and Page Ejection."
.H 4 "Spacing After Headings."
.H 4 "Centered Headings."
.H 4 "Bold, Italic, and Underlined Headings."
.H 5 "Control by Level."
```

4.2.2 Altering Appearance of Headings. Users satisfied with the default appearance of headings may skip to {4.3}. One can modify the appearance of headings quite easily by setting certain registers and strings at the beginning of the document. This permits quick alteration of a document's style, because this style-control information is concentrated in a few lines, rather than being distributed throughout the document.

4.2.2.1 Pre-Spacing and Page Ejection. A first-level heading normally has two blank lines (one vertical space) preceding it, and all others have one blank line (½ a vertical space). If a multi-line heading were to be split across pages, it is automatically moved to the top of the next page. Every first-level heading may be forced to the top of a new page by inserting:

```
.nr Ej 1
```

at the beginning of the document. Long documents may be made more manageable if each section starts on a new page. Setting *Ej* to a higher value causes the same effect for headings up to that level, i.e., a page eject occurs if the heading level is less than or equal to *Ej*.

4.2.2.2 Spacing After Headings. Three registers control the appearance of text immediately following a .H call. They are *Hb* (heading break level), *Hs* (heading space level), and *Hi* (post-heading indent).

If the heading level is less than or equal to *Hb*, a break (3.1) occurs after the heading. If the heading level is less than or equal to *Hs*, a blank line (½ a vertical space) is inserted after the heading. Defaults for *Hb* and *Hs* are 2. If a heading level is greater than *Hb* and also greater than *Hs*, then the heading (if any) is run into the following text. These registers permit headings to be separated from the text in a consistent way throughout a document, while allowing easy alteration of white space and heading emphasis.

For any *stand-alone* heading, i.e., a heading not run into the following text, the alignment of the next line of output is controlled by the register *Hi*. If *Hi* is 0, text is left-justified. If *Hi* is 1 (the *default* value), the text is indented according to the paragraph type as specified by the register *Pt* (4.1). Finally, if *Hi* is 2, text is indented to line up with the first word of the heading itself, so that the heading number stands out more clearly.

For example, to cause a blank line (½ a vertical space) to appear after the first three heading levels, to have no run-in headings, and to force the text following all headings to be left-justified (regardless of the value of *Pt*), the following should appear at the top of the document:

```
.nr Hs 3
.nr Hb 7
.nr Hi 0
```

4.2.2.3 Centered Headings. The register *Hc* can be used to obtain centered headings. A heading is centered if its level is less than or equal to *Hc*, *and* if it is also stand-alone (4.2.2.2). *Hc* is 0 initially (no centered headings).

4.2.2.4 Bold, Italic, and Underlined Headings.

4.2.2.4.1 Control by Level. Any heading that is underlined by *nroff* is made bold or italic by *troff*. The string *HF* (heading font) contains seven codes that specify the fonts for heading levels 1-7. The legal codes, their interpretations, and the defaults for *HF* are:

Formatter	*HF Code*			*Default*
	1	*2*	*3*	*HF*
nroff	no underline	underline	underline	3 3 2 2 2 2 2
troff	roman	italic	bold	3 3 2 2 2 2 2

Thus, all levels are underlined in *nroff*; in *troff*, levels 1 and 2 are bold, levels 3 through 7 are italic. The user may reset HF as desired. Any value omitted from the right end of the list is taken to be 1. For example, the following would result in five underlined (bold) levels and two non-underlined (roman) levels:

```
.ds HF 3 3 3 3 3
```

4.2.2.4.2 Nroff Underlining Style. *Nroff* can underline in two ways. The normal style (.ul request) is to underline only letters and digits. The continuous style (.cu request) underlines all characters, including spaces. By default, PWB/MM attempts to use the continuous style on any heading that is to be underlined and is short enough to fit on a single line. If a heading is to be underlined, but is too long, it is underlined the normal way (i.e., only letters and digits are underlined).

All underlining of headings can be forced to the normal way by using the -rU1 flag when invoking *nroff* (2.4).

4.2.2.4.3 Heading Point Sizes. The user may also specify the desired point size for each heading level with the *HP* string (for use with *troff* only).

```
.ds HP [ps1] [ps2] [ps3] [ps4] [ps5] [ps6] [ps7]
```

By default, the text of headings (.H and .HU) is printed in the same point size as the body *except* that bold stand-alone headings are printed in a size one point smaller than the body. The string *HP*, similar to the string *HF*, can be specified to contain up to seven values, corresponding to the seven levels of headings. For example:

```
.ds HP 12 12 11 10 10 10 10
```

specifies that the first and second level headings are to be printed in 12-point type, with the remainder printed in 10-point. Note that the specified values may also be *relative* point-size changes, e.g.:

```
.ds HP +2 +2 -1 -1
```

If absolute point sizes are specified, then those sizes will be used regardless of the point size of the body of the document. If relative point sizes are specified, then the point sizes for the headings will be relative to the point size of the body, even if the latter is changed.

Omitted or zero values imply that the *default* point size will be used for the corresponding heading level.

☛ *Only the point size of the headings is affected. Specifying a large point size without providing increased vertical spacing (via .HX and/or .HZ) may cause overprinting.*

4.2.2.5 Marking Styles—Numerals and Concatenation.

```
.HM [arg1] ... [arg7]
```

The registers named *H1* through *H7* are used as counters for the seven levels of headings. Their values are normally printed using Arabic numerals. The .HM macro (heading mark style) allows this choice to be overridden, thus providing "outline" and other document styles. This macro can have up to seven arguments; each argument is a string indicating the type of marking to be used. Legal values and their meanings are shown below; omitted values are interpreted as 1, while illegal values have no effect.

Value	*Interpretation*
1	Arabic (default for all levels)
0001	Arabic with enough leading zeroes to get the specified number of digits
A	Upper-case alphabetic
a	Lower-case alphabetic
I	Upper-case Roman
i	Lower-case Roman

By default, the complete heading mark for a given level is built by concatenating the mark for that level to the right of all marks for all levels of higher value. To inhibit the concatenation of heading level marks, i.e., to obtain just the current level mark followed by a period, set the register *Ht* (heading-mark type) to 1.

For example, a commonly-used "outline" style is obtained by:

```
.HM I A 1 a i
.nr Ht 1
```

4.3 Unnumbered Headings

```
.HU heading-text
```

.HU is a special case of .H; it is handled in the same way as .H, except that no heading mark is printed. In order to preserve the hierarchical structure of headings when .H and .HU calls are intermixed, each .HU heading is considered to exist at the level given by register *Hu*, whose initial value is 2. Thus, in the normal case, the only difference between:

.HU heading-text

and

.H 2 heading-text

is the printing of the heading mark for the latter. Both have the effect of incrementing the numbering counter for level 2, and resetting to zero the counters for levels 3 through 7. Typically, the value of *Hu* should be set to make unnumbered headings (if any) be the lowest-level headings in a document.

.HU can be especially helpful in setting up Appendices and other sections that may not fit well into the numbering scheme of the main body of a document (13.2.1).

4.4 Headings and the Table of Contents

The text of headings and their corresponding page numbers can be automatically collected for a table of contents. This is accomplished by doing the following three things:

- specifying in the register *Cl* what level headings are to be saved;
- invoking the .TC macro (10.1) at the end of the document;
- and specifying -rB*n* (2.4) on the command line.

Any heading whose level is less than or equal to the value of the register *Cl* (contents level) is saved and later displayed in the table of contents. The default value for *Cl* is 2, i.e., the first two levels of headings are saved.

Due to the way the headings are saved, it is possible to exceed the formatter's storage capacity, particularly when saving many levels of many headings, while also processing displays (7) and footnotes (8). If this happens, the "Out of temp file space" diagnostic (Appendix E) will be issued; the only remedy is to save fewer levels and/or to have fewer words in the heading text.

4.5 First-Level Headings and the Page Numbering Style

By default, pages are numbered sequentially at the top of the page. For large documents, it may be desirable to use page numbering of the form "section-page," where *section* is the number of the current first-level heading. This page numbering style can be achieved by specifying the flag -rN3 on the command line (9.9). As a side effect, this also has the effect of setting *Ej* to 1, i.e., each section begins on a new page. In this style, the page number is printed at the *bottom* of the page, so that the correct section number is printed.

4.6 User Exit Macros •

☛ *This section is intended only for users who are accustomed to writing formatter macros.*

.HX dlevel rlevel heading-text
.HZ dlevel rlevel heading-text

The .HX and .HZ macros are the means by which the user obtains a final level of control over the previously-described heading mechanism. PWB/MM does not define .HX and .HZ; they are intended to be defined by the user. The .H macro invokes .HX shortly before the actual heading text is printed; it calls .HZ as its last action. All the default actions occur if these macros are not defined. If the .HX or .HZ (or both) are defined by the user, the user-supplied definition is interpreted at the appropriate point. These macros can therefore influence the handling of all headings, because the .HU macro is actually a special case of the .H macro.

If the user originally invoked the .H macro, then the derived level (*dlevel*) and the real level (*rlevel*) are both equal to the level given in the .H invocation. If the user originally invoked the .HU macro (4.3), *dlevel* is equal to the contents of register *Hu*, and *rlevel* is 0. In both cases, *heading-text* is the text of the original invocation.

By the time .H calls .HX, it has already incremented the heading counter of the specified level (4.2.2.5), produced blank line(s) (vertical space) to precede the heading (4.2.2.1), and accumulated the

"heading mark", i.e., the string of digits, letters, and periods needed for a numbered heading. When .HX is called, all user-accessible registers and strings can be referenced, as well as the following:

string }0 — If *rlevel* is non-zero, this string contains the "heading mark." Two unpaddable spaces (to separate the *mark* from the *heading*) have been appended to this string. If *rlevel* is 0, this string is null.

register ;0 — This register indicates the type of spacing that is to follow the heading {4.2.2.2}. A value of 0 means that the heading is run-in. A value of 1 means a break (but no blank line) is to follow the heading. A value of 2 means that a blank line (½ a vertical space) is to follow the heading.

string }2 — If register ;0 is 0, this string contains two unpaddable spaces that will be used to separate the (run-in) *heading* from the following *text*. If register ;0 is non-zero, this string is null.

register ;3 — This register contains an adjustment factor for a .ne request issued before the heading is actually printed. On entry to .HX, it has the value 3 if *dlevel* equals 1, and 1 otherwise. The .ne request is for the following number of lines: the contents of the register ;0 taken as blank lines (halves of vertical space) plus the contents of register ;3 as blank lines (halves of vertical space) plus the number of lines of the heading.

The user may alter the values of *}0*, *}2*, and *;3* within .HX as desired. The following are examples of actions that might be performed by defining .HX to include the lines shown:

Change first-level heading mark from format *n.* to *n.*0:
```
.if \\$1=1 .ds }0 \\n(H1.0\□\□
```
(□ stands for a space)

Separate run-in heading from the text with a period and two unpaddable spaces:
```
.if \\n(;0=0 .ds }2 .\□\□
```

Assure that at least 15 lines are left on the page before printing a first-level heading:
```
.if \\$1=1 .nr ;3 15-\\n(;0
```

Add 3 additional blank lines before each first-level heading:
```
.if \\$1=1 .sp 3
```

If temporary string or macro names are used within .HX, care must be taken in the choice of their names {13.1}.

.HZ is called at the end of .H to permit user-controlled actions after the heading is produced. For example, in a large document, sections may correspond to chapters of a book, and the user may want to reset counters for footnotes, figures, tables, etc. Another use might be to change a page header or footer. For example:

```
.de HZ
.if \\$1=1 \{.nr :p 0 \" footnotes
.      nr Fg 0 \" figures
.      nr Tb 0 \" tables
.      nr Ec 0 \" equations
.      PF "'Section \\$3'"\}
..
```

4.7 Hints for Large Documents

A large document is often organized for convenience into one file per section. If the files are numbered, it is wise to use enough digits in the names of these files for the maximum number of sections, i.e., use suffix numbers 01 through 20 rather than 1 through 9 and 10 through 20.

Users often want to format individual sections of long documents. To do this with the correct section numbers, it is necessary to set register *H1* to 1 less than the number of the section just *before* the

corresponding ".H 1" call. For example, at the beginning of section 5, insert:

```
.nr H1 4
```

☛ *This is a dangerous practice: it defeats the automatic (re)numbering of sections when sections are added or deleted. Remove such lines as soon as possible.*

5. LISTS

This section describes many different kinds of lists: automatically-numbered and alphabetized lists, bullet lists, dash lists, lists with arbitrary marks, and lists starting with arbitrary strings, e.g., with terms or phrases to be defined.

5.1 Basic Approach

In order to avoid repetitive typing of arguments to describe the appearance of items in a list, PWB/MM provides a convenient way to specify lists. All lists are composed of the following parts:

- A *list-initialization* macro that controls the appearance of the list: line spacing, indentation, marking with special symbols, and numbering or alphabetizing.
- One or more *List Item* (.LI) macros, each followed by the actual text of the corresponding list item.
- The *List End* (.LE) macro that terminates the list and restores the previous indentation.

Lists may be nested up to five levels. The list-initialization macro saves the previous list status (indentation, marking style, etc.); the .LE macro restores it.

With this approach, the format of a list is specified only once at the beginning of that list. In addition, by building on the existing structure, users may create their own customized sets of list macros with relatively little effort (5.4, Appendix A, Appendix B).

5.2 Sample Nested Lists

The input for several lists and the corresponding output are shown below. The .AL and .DL macro calls (5.3.3) contained therein are examples of the *list-initialization* macros. This example will help us to explain the material in the following sections. Input text:

```
.AL A
.LI
This is an alphabetized item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.AL
.LI
This is a numbered item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.DL
.LI
This is a dash item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LI + 1
This is a dash item with a "plus" as prefix.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LE
.LI
This is numbered item 2.
.LE
.LI
This is another alphabetized item, B.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LE
.P
This paragraph appears at the left margin.
```

Output:

A. This is an alphabetized item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

    1. This is a numbered item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

        — This is a dash item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

    + — This is a dash item with a "plus" as prefix. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

    2. This is numbered item 2.

B. This is another alphabetized item, B. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

This paragraph appears at the left margin.

5.3 Basic List Macros

Because all lists share the same overall structure except for the list-initialization macro, we first discuss the macros common to all lists. Each list-initialization macro is covered in {5.3.3}.

5.3.1 List Item.

.LI [mark] [1]
one or more lines of text that make up the list item.

The .LI macro is used with all lists. It normally causes the output of a single blank line (½ a vertical space) before its item, although this may be suppressed. If no arguments are given, it labels its item with the *current mark*, which is specified by the most recent list-initialization macro. If a single argument is given to .LI, that argument is output *instead of* the current mark. If two arguments are given, the first argument becomes a *prefix* to the current mark, thus allowing the user to emphasize one or more items in a list. One unpaddable space is inserted between the prefix and the mark. For example:

```
.BL 6
.LI
This is a simple bullet item.
.LI +
This replaces the bullet with a "plus."
.LI + xxx
But this uses "plus" as prefix to the bullet.
.LE
```

yields:

- This is a simple bullet item.

+ This replaces the bullet with a "plus."

\+ • But this uses "plus" as prefix to the bullet.

☞ *The* mark *must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified* {3.3}.

If the *current mark* (in the *current list*) is a null string, and the first argument of .LI is omitted or null, the resulting effect is that of a *hanging indent*, i.e., the first line of the following text is "outdented," starting at the same place where the *mark* would have started {5.3.3.6}.

5.3.2 List End.

.LE [1]

List End restores the state of the list back to that existing just before the most recent list-initialization macro call. If the optional argument is given, the .LE outputs a blank line (½ a vertical space). This option should generally be used only when the .LE is followed by running text, but not when followed by a macro that produces blank lines of its own, such as .P, .H, or .LI.

.H and .HU automatically clear all list information, so one may legally omit the .LE(s) that would normally occur just before either of these macros. Such a practice is *not* recommended; however, because errors will occur if the list text is separated from the heading at some later time (e.g., by insertion of text).

5.3.3 List Initialization Macros. The following are the various list-initialization macros. They are actually implemented as calls to the more basic .LB macro {5.4}.

5.3.3.1 Automatically-Numbered or Alphabetized Lists.

.AL [type] [text-indent] [1]

The .AL macro is used to begin sequentially-numbered or alphabetized lists. If there are no arguments, the list is numbered, and text is indented *Li* (initially 5)[4] spaces from the indent in force when the .AL

is called, thus leaving room for two digits, a period, and two spaces before the text.

Spacing at the beginning of the list and between the items can be suppressed by setting the *Ls* (list space) register. *Ls* is set to the innermost list level for which spacing *is* done. For example:

```
.nr Ls 0
```

specifies that no spacing will occur around *any* list items. The default value for *Ls* is 6 (which is the *maximum* list nesting level).

The *type* argument may be given to obtain a different type of sequencing, and its value should indicate the first element in the sequence desired, i.e., it must be 1, A, a, I, or i {4.2.2.5}.[5] If *type* is omitted or null, then "1" is assumed. If *text-indent* is non-null, it is used as the number of spaces from the current indent to the text, i.e., it is used instead of *Li* for this list only. If *text-indent* is null, then the value of *Li* will be used.

If the third argument is given, a blank line (½ a vertical space) will *not* separate the items in the list. A blank line (½ a vertical space) will occur before the first item, however.

5.3.3.2 Bullet List

```
.BL [text-indent] [1]
```

.BL begins a bullet list, in which each item is marked by a bullet (•) followed by one space. If *text-indent* is non-null, it overrides the default indentation—the amount of paragraph indentation as given in the register *Pi* {4.1}.[6]

If a second argument is specified, no blank lines will separate the items in the list.

5.3.3.3 Dash List

```
.DL [text-indent] [1]
```

.DL is identical to .BL, except that a dash is used instead of a bullet.

5.3.3.4 Marked List

```
.ML mark [text-indent] [1]
```

.ML is much like .BL and .DL, but expects the user to specify an arbitrary mark, which may consist of more than a single character. Text is indented *text-indent* spaces if the second argument is not null; otherwise, the text is indented one more space than the width of *mark*. If the third argument is specified, no blank lines will separate the items in the list.

☞ *The* mark *must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified* {3.3}.

5.3.3.5 Reference List

```
.RL [text-indent] [1]
```

A .RL call begins an automatically-numbered list in which the numbers are enclosed by square brackets ([]). *Text-indent* may be supplied, as for .AL. If omitted or null, it is assumed to be 6, a convenient value for lists numbered up to 99. If the second argument is specified, no blank lines will separate the items in the list. The list of references {14} was produced using the .RL macro.

---

4. Values that specify indentation must be *unscaled* and are treated as "character positions," i.e., as the number of *ens*.
5. Note that the "0001" format is *not* permitted.
6. So that, in the default case, the text of bullet and dash lists lines up with the first line of indented paragraphs.

5.3.3.6 Variable-Item List

.VL text-indent [mark-indent] [1]

When a list begins with a .VL, there is effectively no *current mark;* it is expected that each .LI will provide its own mark. This form is typically used to display definitions of terms or phrases. *Mark-indent* gives the number of spaces from the current indent to the beginning of the *mark*, and it defaults to 0 if omitted or null. *Text-indent* gives the distance from the current indent to the beginning of the text. If the third argument is specified, no blank lines will separate the items in the list. Here is an example of .VL usage:

```
.ta ~
.VL 20 2
.LI mark~1
Here is a description of mark 1:
"mark 1" of the .LI line contains a tilde translated to an unpaddable space in order
to avoid extra spaces between
"mark" and "1" (3.3).
.LI second~mark
This is the second mark, also using a tilde translated to an unpaddable space.
.LI third~mark~longer~than~indent:
This item shows the effect of a long mark; one space separates the mark
from the text.
.LI ~
This item effectively has no mark because the
tilde following the .LI is translated into a space.
.LE
```

yields:

mark 1 — Here is a description of mark 1; "mark 1" of the .LI line contains a tilde translated to an unpaddable space in order to avoid extra spaces between "mark" and "1" (3.3).

second mark — This is the second mark, also using a tilde translated to an unpaddable space.

third mark longer than indent: — This item shows the effect of a long mark; one space separates the mark from the text.

This item effectively has no mark because the tilde following the .LI is translated into a space.

The tilde argument on the last .LI above is required; otherwise a *hanging indent* would have been produced. A *hanging indent* is produced by using .VL and calling .LI with no arguments or with a null first argument. For example:

```
.VL 10
.LI
Here is some text to show a hanging indent.
The first line of text is at the left margin.
The second is indented 10 spaces.
.LE
```

yields:

Here is some text to show a hanging indent. The first line of text is at the left margin. The second is indented 10 spaces.

☛ *The* mark *must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified* (3.3).

5.4 List-Begin Macro and Customized Lists •

.LB text-indent mark-indent pad type [mark] [LI-space] [LB-space]

The list-initialization macros described above suffice for almost all cases. However, if necessary, one may obtain more control over the layout of lists by using the basic list-begin macro .LB, which is also used by all the other list-initialization macros (Appendix A). Its arguments are as follows:

Text-indent gives the number of spaces that the text is to be indented from the current indent. Normally, this value is taken from the register *Li* for automatic lists and from the register *Pi* for bullet and dash lists.

The combination of *mark-indent* and *pad* determines the placement of the mark. The mark is placed within an area (called *mark area*) that starts *mark-indent* spaces to the right of the current indent, and ends where the text begins (i.e., ends *text-indent* spaces to the right of the current indent).[7] Within the mark area, the mark is *left-justified* if *pad* is 0. If *pad* is greater than 0, say *n*, then *n* blanks are appended to the mark; the *mark-indent* value is ignored. The resulting string immediately precedes the text. That is, the mark is effectively *right-justified pad* spaces immediately to the left of the text.

Type and *mark* interact to control the type of marking used. If *type* is 0, simple marking is performed using the mark character(s) found in the *mark* argument. If *type* is greater than 0, automatic numbering or alphabetizing is done, and *mark* is then interpreted as the first item in the sequence to be used for numbering or alphabetizing, i.e., it is chosen from the set (1, A, a, I, i) as in (5.3.3.1). That is:

Type	*Mark*	*Result*
0	omitted	hanging indent
0	*string*	*string* is the mark
>0	omitted	arabic numbering
>0	one of: 1, A, a, I, i	automatic numbering or alphabetic sequencing

Each non-zero value of *type* from 1 to 6 selects a different way of displaying the items. The following table shows the output appearance for each value of *type:*

Type	*Appearance*
1	x.
2	x)
3	(x)
4	[x]
5	<x>
6	{x}

where *x* is the generated number or letter.

☛ *The* mark *must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified* (3.3).

LI-space gives the number of blank lines (halves of a vertical space) that should be output by each .LI macro in the list. If omitted, *LI-space* defaults to 1; the value 0 can be used to obtain compact lists. If *LI-space* is greater than 0, the .LI macro issues a .ne request for two lines just before printing the mark.

LB-space, the number of blank lines (½ a vertical space) to be output by .LB itself, defaults to 0 if omitted.

---

7. The *mark-indent* argument is typically 0.

There are three reasonable combinations of *LI-space* and *LB-space*. The normal case is to set *LI-space* to 1 and *LB-space* to 0, yielding one blank line *before* each item in the list; such a list is usually terminated with a ".LE 1" to end the list with a blank line. In the second case, for a more compact list, set *LI-space* to 0 and *LB-space* to 1, and, again, use ".LE 1" at the end of the list. The result is a list with one blank line before and after it. If you set both *LI-space* and *LB-space* to 0, and use ".LE" to end the list, a list without *any* blank lines will result.

Appendix A shows the definitions of the list-initialization macros (5.3.3) in terms of the .LB macro. Appendix B illustrates how the user can build upon those macros to obtain other kinds of lists.

6. MEMORANDUM AND RELEASED PAPER STYLES

One use of PWB/MM is for the preparation of memoranda and released papers, which have special requirements for the first page and for the cover sheet. The information needed for the memorandum or released paper (title, author, date, case numbers, etc.) is entered in the same way for *both* styles; an argument to one macro indicates which style is being used. The following sections describe the macros used to provide this data. The required order is shown in (6.9).

If neither the memorandum nor released-paper style is desired, the macros described below should be omitted from the input text. If these macros are omitted, the first page will simply have the page header (9) followed by the body of the document.

6.1 Title

```
.TL [charging-case] [filing-case]
one or more lines of title text
```

The arguments to the .TL macro are the charging case number(s) and filing case number(s).[8] The title of the memorandum or paper follows the .TL macro and is processed in fill mode (3.1). Multiple charging case numbers are entered as "sub-arguments" by separating each from the previous with a comma and a space, and enclosing the *entire* argument within double quotes. Multiple filing case numbers are entered similarly. For example:

```
.TL "12345, 67890" 987654321
On the construction of a table
of all even prime numbers
```

The .br request may be used to break the title into several lines.

On output, the title appears after the word "subject" in the memorandum style. In the released-paper style, the title is centered and underlined (bold).

6.2 Author(s)

```
.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]
```

The .AU macro receives as arguments information that describes an author. If any argument contains blanks, it must be enclosed within double quotes. The first six arguments must appear in the order given (a separate .AU macro is required for each author). For example:

```
.AU "J. J. Jones" JJJ PY 9876 5432 1Z-234
```

In the "from" portion in the memorandum style, the author's name is followed by location and department number on one line and by room number and extension number on the next. The "x" for the extension is added automatically. The printing of the location, department number, extension

8. The "charging case" is the case number to which time was charged for the development of the project described in the memorandum. The "filing case" is a number under which the memorandum is to be filed.

number, and room number may be suppressed on the first page of a memorandum by setting the register *Au* to 0; the default value for *Au* is 1. Arguments 7 through 9, if present, will follow this "normal" author information, each on a separate line. Certain organizations have their own numbering schemes for memoranda, engineer's notes, etc. These numbers are printed after the author's name. This can be done by providing more than six arguments to the .AU macro, e.g.:

```
.AU "S. P. Lename" SPL IH 9988 7766 5H-444 3322.11AB
```

The name, initials, location, and department are also used in the Signature Block (6.11.1). The author information in the "from" portion, as well as the names and initials in the Signature Block will appear in the same order as the .AU macros.

The names of the authors in the released-paper style are centered below the title. After the name of the last author, "Bell Laboratories" and the location are centered. For the case of authors from different locations, see (6.8).

6.3 TM Number(s)

```
.TM [number] ...
```

If the memorandum is a Technical Memorandum, the TM numbers are supplied via the .TM macro. Up to nine numbers may be specified. Example:

```
.TM 7654321 77777777
```

This macro call is ignored in the released-paper and external-letter styles (6.6).

6.4 Abstract

```
.AS [arg] [indent]
text of the abstract
.AE
```

In both the memorandum and released-paper styles, the text of the abstract follows the author information and is preceded by the centered and underlined (italic) word "ABSTRACT."

The .AS (abstract start) and .AE (abstract end) macros bracket the (optional) abstract. The first argument to .AS controls the printing of the abstract. If it is 0 or null, the abstract is printed on the first page of the document, immediately following the author information, and is also saved for the cover sheet. If the first argument is 1, the abstract is saved and printed only on the cover sheet. The margins of the abstract are indented on the left and right by five spaces. The amount of indentation can be changed by specifying the desired indentation as the second argument.[9]

Note that headings (4.2, 4.3) and displays (7) are *not* (as yet) permitted within an abstract.

6.5 Other Keywords

```
.OK [keyword] ...
```

Topical keywords should be specified on a Technical Memorandum cover sheet. Up to nine such keywords or keyword phrases may be specified as arguments to the .OK macro; if any keyword contains spaces, it must be enclosed within double quotes.

6.6 Memorandum Types

```
.MT [type] [1]
```

---

9. Values that specify indentation must be *unscaled* and are treated as "character positions," i.e., as the number of *ems*.

The .MT macro controls the format of the top part of the first page of a memorandum or of a released paper, as well as the format of the cover sheets. Legal codes for *type* and the corresponding values are:

Code	*Value*
.MT ""	no memorandum type is printed
.MT 0	no memorandum type is printed
.MT	MEMORANDUM FOR FILE
.MT 1	MEMORANDUM FOR FILE
.MT 2	PROGRAMMER'S NOTES
.MT 3	ENGINEER'S NOTES
.MT 4	Released-Paper style
.MT 5	External-Letter style
.MT "*string*"	*string*

If *type* indicates a memorandum style, then *value* will be printed after the last line of author information or after the last line of the abstract, if one appears on the first page. If *type* is longer than one character, then the string, itself, will be printed. For example:

.MT "Technical Note #5"

A simple letter is produced by calling .MT with a null (but *not* omitted!) or zero argument.

The second argument to .MT is used only if the first argument is 4 (i.e., for the released-paper style) as explained in [6.8].

In the external-letter style (.MT 5), only the date is printed in the upper right corner of the first page. It is expected that preprinted stationery will be used, providing the author's company logotype and address.

6.7 Date and Format Changes

6.7.1 Changing the Date. By default, the current date appears in the "date" part of a memorandum. This can be overridden by using:

.ND new-date

The .ND macro alters the value of the string *DT*, which is initially set to the current date.

6.7.2 Alternate First-Page Format. One can specify that the words "subject," "date," and "from" (in the memorandum style) be omitted and that an alternate company name be used:

.AF [company-name]

If an argument is given, it replaces "Bell Laboratories", without affecting the other headings. If the argument is *null*, "Bell Laboratories" is suppressed; in this case, extra blank lines are inserted to allow room for stamping the document with a Bell System logo or a Bell Laboratories stamp. .AF with *no* argument suppresses "Bell Laboratories" and the "Subject/Date/From" *headings*, thus allowing output on preprinted stationery.

The only .AF option appropriate for *troff* is to specify an argument to replace "Bell Laboratories" with another name.

6.8 Released-Paper Style

The released-paper style is obtained by specifying:

.MT 4 [1]

This results in a centered, underlined (bold) title followed by centered names of authors. The location of the last author is used as the location following "Bell Laboratories" (unless .AF [6.7.2] specifies a different company). If the optional second argument to .MT is given, then the name of each author is followed by the respective company name and location. The abstract, if present, follows the author

information.

Information necessary for the memorandum style but not for the released-paper style is ignored.

If the released-paper style is utilized, most BTL location codes[10] are defined as strings that are the addresses of the corresponding BTL locations. These codes are needed only until the .MT macro is invoked. Thus, *following* the .MT macro, the user may re-use these string names. In addition, the macros described in {6.11} and their associated lines of input are ignored when the released-paper style is specified.

Authors from non-BTL locations may include their affiliations in the released-paper style by specifying the appropriate .AF *before* each .AU. For example:

```
.TL
A Learned Treatise
.AF "Getem Inc."
.AU "F. Swatter"
.AF "Bell Laboratories"
.AU "Sam P. Lename" "" CB
.MT 4 1
```

6.9 Order of Invocation of "Beginning" Macros

The macros described in {6.1-6.7}, *if present*, must be given in the following order:

```
.ND new-date
.TL [charging-case] [filing-case]
one or more lines of text
.AF [company-name]
.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]
.TM [number] ...
.AS [arg] [indent]
one or more lines of text
.AE
.OK [keyword] ...
.MT [type] [1]
```

The only *required* macros for a memorandum or a released paper are .TL, .AU, and .MT; all the others (and their associated input lines) may be omitted if the features they provide are not needed. Once .MT has been invoked, *none* of the above macros can be re-invoked because they are removed from the table of defined macros to save space.

6.10 Example

The input text for this manual begins as follows:

```
.TL
P\s-3WB/MM\s0\(emProgrammer's Workbench Memorandum Macros
.AU "D. W. Smith" DWS PY ...
.AU "J. R. Mashey" JRM  MH ...
.MT 4 1
```

10. The complete list is: AK, CP, CH, CB, DR, HO, IN, IH, MV, MH, PY, RR, RD, WV, and WH.

6.11 Macros for the End of a Memorandum

At the end of a memorandum (but not of a released paper), the signatures of the authors and a list of notations[11] can be requested. The following macros and their input are ignored if the released-paper style is selected.

6.11.1 Signature Block

.SG [arg] [1]

.SG prints the author name(s) after the last line of text, aligned with the "Date/From" block. Three blank lines are left above each name for the actual signature. If no argument is given, the line of reference data[12] will *not* appear following the last line.

A non-null first argument is treated as the typist's initials, and is appended to the reference data. Supply a null argument to print reference data with neither the typist's initials nor the preceding hyphen.

If there are several authors and if the second argument is given, then the reference data is placed on the same line as the name of the first author, rather than on the line that has the name of the last author.

The reference data contains only the location and department number of the first author. Thus, if there are authors from different departments and/or from different locations, the reference data should be supplied manually after the invocation (without arguments) of the .SG macro. For example:

```
.SG
.rs
.sp -1v
PY/MH-9876/5432-JJJ/SPL-cen
```

6.11.2 "Copy to" and Other Notations

```
.NS [arg]
zero or more lines of the notation
.NE
```

After the signature and reference data, many types of notations may follow, such as a list of attachments or "copy to" lists. The various notations are obtained through the .NS macro, which provides for the proper spacing and for breaking the notations across pages, if necessary.

The codes for *arg* and the corresponding notations are:

---

11. See [2], pp. 1.12–16

12. The following information is known as reference data: location code, department number, author's initials, and typist's initials, all separated by hyphens. See [2], page 1.11

Code	*Notations*
.NS ""	Copy to
.NS 0	Copy to
.NS	Copy to
.NS 1	Copy (with att.) to
.NS 2	Copy (without att.) to
.NS 3	Att.
.NS 4	Atts.
.NS 5	Enc.
.NS 6	Encs.
.NS 7	Under Separate Cover
.NS 8	Letter to
.NS 9	Memorandum to
.NS "*string*"	Copy (*string*) to

If *arg* consists of more than one character, it is placed within parentheses between the words "Copy" and "to." For example:

```
.NS "with att. 1 only"
```

will generate "Copy (with att. 1 only) to" as the notation. More than one notation may be specified before the .NE occurs, because a .NS macro terminates the preceding notation, if any. For example:

```
.NS 4
Attachment 1-List of register names
Attachment 2-List of string and macro names
.NS 1
J. J. Jones
.NS 2
S. P. Lename
G. H. Hurtz
.NE
```

would be formatted as:

Atts.
Attachment 1-List of register names
Attachment 2-List of string and macro names

Copy (with att.) to
J. J. Jones

Copy (without att.) to
S. P. Lename
G. H. Hurtz

6.12 Forcing a One-Page Letter

At times, one would like just a bit more space on the page, forcing the signature or items within notations onto the bottom of the page, so that the letter or memo is just one page in length. This can be accomplished by increasing the page length through the -rL*n* option, e.g. -rL90. This has the effect of making the formatter believe that the page is 90 lines long and therefore giving it more room than usual to place the signature or the notations. This will *only* work for a *single-page* letter or memo.

7. DISPLAYS

Displays are blocks of text that are to be kept together—not split across pages. PWB/MM provides two styles of displays:[13] a *static* (.DS) style and a *floating* (.DF) style. In the *static* style, the display appears

in the same relative position in the output text as it does in the input text; this may result in extra white space at the bottom of the page if the display is too big to fit there. In the *floating* style, the display "floats" through the input text to the top of the next page if there is not enough room for it on the current page; thus the input text that *follows* a floating display may *precede* it in the output text. A queue of floating displays is maintained so that their relative order is not disturbed.

By default, a display is processed in no-fill mode and is *not* indented from the existing margin. The user can specify indentation or centering, as well as fill-mode processing.

Displays and footnotes [8] may *never* be nested, in any combination whatsoever. Although lists [5] and paragraphs [4.1] are permitted, no headings (.H or .HU) can occur within displays or footnotes.

7.1 Static Displays

```
.DS [format] [fill]
one or more lines of text
.DE
```

A static display is started by the .DS macro and terminated by the .DE macro. With no arguments, .DS will accept the lines of text exactly as they are typed (no-fill mode) and will *not* indent them from the prevailing indentation. The *format* argument to .DS is an integer or letter with the following meanings:

Code	*Meaning*
""	no indent
0 or L	no indent
1 or I	indent by standard amount
2 or C	center each line
3 or CB	center as a block

The *fill* argument is also an integer or letter and can have the following meanings:

Code	*Meaning*
""	no-fill mode
0 or N	no-fill mode
1 or F	fill mode

Omitted arguments are taken to be zero.

The standard amount of indentation is taken from the register *Si*, which is initially 5. Thus, by default, the text of an indented display aligns with the first line of indented paragraphs, whose indent is contained in the *Pi* register [4.1]. Even though their initial values are the same, these two registers are independent of one another.

The display format value 3 (CB) centers the *entire display* as a block (as opposed to .DS 2 and .DF 2, which center each line *individually*). That is, all the collected lines are left-justified, and then the display is centered, based on the width of the longest line. This format *must* be used in order for the *eqn*(1)/*neqn*(1) "mark" and "lineup" feature to work with centered equations (see section 7.4 below).

By default, a blank line (½ a vertical space) is placed before and after static and floating displays. These blank lines before and after *static* displays can be inhibited by setting the register *Ds* to 0.

---

13. Displays are processed in an environment that is different from that of the body of the text (see the .ev request in [9]).

7.2 Floating Displays

```
.DF [format] [fill]
one or more lines of text
.DE
```

A floating display is started by the .DF macro and terminated by the .DE macro. The arguments have the same meanings as for .DS [7.1], except that, for floating displays, indent, no indent, and centering are always calculated with respect to the initial left margin, because the prevailing indent may change between the time when the formatter first reads the floating display and the time that the display is printed. One blank line (½ a vertical space) *always* occurs both before and after a floating display.

The user may exercise great control over the output positioning of floating displays through the use of two number registers, *De* and *Df*. When a floating display is encountered by *nroff* or *troff*, it is processed and placed onto a queue of displays waiting to be output. Displays are always removed from the queue and printed in the order that they were entered on the queue, which is the order that they appeared in the input file. If a new floating display is encountered and the queue of displays is empty, then the new display is a candidate for immediate output on the current page. Immediate output is governed by the size of the display and the setting of the *Df* register (see below). The *De* register (see below) controls whether or not text will appear on the current page after a floating display has been produced.

As long as the queue contains one or more displays, new displays will be automatically entered there, rather than being output. When a new page is started (or the top of the second column when in two-column mode) the next display from the queue becomes a candidate for output if the *Df* register has specified "top-of-page" output. When a display is output it is also removed from the queue.

When the end of a section (when using section-page numbering) or the end of a document is reached, all displays are automatically removed from the queue and output. This will occur before a .CS or .TC is processed.

A display is said to "fit on the current page" if there is enough room to contain the entire display on the page, or if the display is longer than one page in length and less than half of the current page has been used. Also note that a wide (full page width) display will never fit in the second column of a two-column document.

The registers, their settings, and their effects are as follows:

Values for *De* Register	
Value	**Action**
0	DEFAULT: No special action occurs.
1	A page eject will *always* follow the output of each floating display, so only one floating display will appear on a page and no text will follow it. NOTE: For any other values the action performed is for the value 1.

Values for *Df* Register	
Value	**Action**
0	Floating displays will not be output until end of section (when section-page numbering) or end of document.

Values for *Df* Register	
Value	Action
1	Output the new floating display on the current page if there is room, otherwise hold it until the end of the section or document.
2	Output exactly one floating display from the queue at the top of a new page or column (when in two-column mode).
3	Output one floating display on current page if there is room. Output exactly one floating display at the top of a new page or column.
4	Output as many displays as will fit (at least one), starting at the top of a new page or column. Note that if register *De* is set to 1, each display will be followed by a page eject, causing a new top of page to be reached where at least one more display will be output. (This also applies to value 5, below.)
5	DEFAULT: Output a new floating display on the current page if there is room. Output at least one, but as many displays as will fit starting at the top of a new page or column. NOTE: For any value greater than 5 the action performed is for the value 5.

7.3 Tables

```
.TS [H]
global options;
column descriptors.
title lines
[.TH [N]]
data within the table.
.TE
```

The .TS (table start) and .TE (table end) macros make possible the use of the *tbl*(1) processor [11]. They are used to delimit the text to be examined by *tbl*(1) as well as to set proper spacing around the table. The display function and the *tbl*(1) delimiting function are independent of one another, however, so in order to permit one to keep together blocks that contain any mixture of tables, equations, filled and unfilled text, and caption lines the .TS-.TE block should be enclosed within a display (.DS-.DE). Floating tables may be enclosed inside floating displays (.DF-.DE).

The macros .TS and .TE also permit the processing of tables that extend over several pages. If a table heading is needed for each page of a multi-page table, specify the argument "H" to the .TS macro as above. Following the options and format information, the table heading is typed on as many lines as required and followed by the .TH macro. The .TH macro *must* occur when ".TS H" is used. Note that this is *not* a feature of *tbl*(1), but of the macro definitions provided by PWB/MM.

The table header macro .TH may take as an argument the letter N. This argument causes the table header to be printed only if it is the first table header on the page. This option is used when it is necessary to build long tables from smaller .TS H/.TE segments. For example:

```
.TS H
global options;
column descriptors.
Title lines
.TH
data
.TE
.TS H
global options;
column descriptors.
Title lines
.TH N
data
.TE
```

will cause the table heading to appear at the top of the first table segment, and no heading to appear at the top of the second segment when both appear on the same page. However, the heading will still appear at the top of each page that the table continues onto. This feature is used when a single table must be broken into segments because of table complexity (for example, too many blocks of filled text). If each segment had its own .TS H/.TH sequence, each segment would have its own header. However, if each table segment after the first uses .TS H/.TH N then the table header will only appear at the beginning of the table and the top of each new page or column that the table continues onto.

7.4 Equations

```
.DS
.EQ [label]
equation(s)
.EN
.DE
```

The equation setters *eqn*(1) and *neqn*(1) [6,7] expect to use the .EQ (equation start) and .EN (equation end) macros as delimiters in the same way that *tbl*(1) uses .TS and .TE; however, .EQ and .EN must occur inside a .DS-.DE pair.

The .EQ macro takes an argument that will be used as a label for the equation. The label will appear at the right margin in the "vertical center" of the general equation.

☛ *There is an exception to this rule: if .EQ and .EN are used only to specify the delimiters for in-line equations or to specify* eqn/neqn *"defines," .DS and .DE must* not *be used; otherwise extra blank lines will appear in the output.*

7.5 Figure, Table, Equation, and Exhibit Captions

.FG [title] [override] [flag]
.TB [title] [override] [flag]
.EC [title] [override] [flag]
.EX [title] [override] [flag]

The .FG (Figure Title), .TB (Table Title), .EC (Equation Caption) and .EX (Exhibit Caption) macros are normally used inside .DS-.DE pairs to automatically number and title figures, tables, and equations. They use registers *Fg, Tb, Ec,* and *Ex,* respectively.[14] As an example, the call:

.FG "This is an illustration"

yields:

Figure 1. This is an illustration

.TB replaces "Figure" by "TABLE"; .EC replaces "Figure" by "Equation", and .EX replaces "Figure" by "Exhibit". Output is centered if it can fit on a single line; otherwise, all lines but the first are indented to line up with the first character of the title. The format of the numbers may be changed using the .af request of the formatter.

The *override* string is used to modify the normal numbering. If *flag* is omitted or 0, *override* is used as a prefix to the number; if *flag* is 1, *override* is used as a suffix; and if *flag* is 2, *override* replaces the number. For example, to produce figures numbered within sections, supply \n(H1 for *override* on each .FG call, and reset *Fg* at the beginning of each section, as shown in [4.6].

As a matter of style, table headings are usually placed ahead of the text of the tables, while figure, equation, and exhibit captions usually occur after the corresponding figures and equations.

7.6 List of Figures, Tables, Equations, and Exhibits

A List of Figures, List of Tables, List of Exhibits, and List of Equations may be obtained. They will be printed after the Table of Contents is printed if the number registers *Lf, Lt, Lx,* and *Le* (respectively) are set to 1. *Lf, Lt,* and *Lx* are 1 by default; *Le* is 0 by default.

7.7 Blocks of Filled Text

One can obtain blocks of filled text through the use of .DS or .DF. However, to have the block of filled text *centered* within the current line length, the *tbl*(1) program may be used:

14. The user may wish to reset these registers after each first-level heading [4.6].

```
  :
  :
.DS 0 1
.TS
center;
lw40 .
T{
  :
  :
T}
.TE
.DE
  :
  :
```

The ".DS 0 1" begins a non-indented, filled display. The *tbl*(1) parameters set up a centered table with a column width of 40 ens. The "T{ ... T}" sequence allows filled text to be input as data within a table.

8. FOOTNOTES

There are two macros that delimit the text of footnotes,[15] a string used to automatically number the footnotes, and a macro that specifies the style of the footnote text.

8.1 Automatic Numbering of Footnotes

Footnotes may be automatically numbered by typing the three characters "*F" immediately after the text to be footnoted, without any intervening spaces. This will place the next sequential footnote number (in a smaller point size) a half-line above the text to be footnoted.

8.2 Delimiting Footnote Text

There are two macros that delimit the text of each footnote:

```
.FS [label]
one or more lines of footnote text
.FE
```

The .FS (footnote start) marks the beginning of the text of the footnote, and the .FE marks its end. The *label* on the .FS, if present, will be used to mark the footnote text. Otherwise, the number retrieved from the string F will be used. Note that automatically-numbered and user-labeled footnotes may be intermixed. If a footnote is labeled (.FS *label*), the text to be footnoted *must* be followed by *label*, rather than by "*F". The text between .FS and .FE is processed in fill mode. Another .FS, a .DS, or a .DF are *not* permitted between the .FS and .FE macros. Examples:

1. Automatically-numbered footnote:

```
This is the line containing the word\*F
.FS
This is the text of the footnote.
.FE
to be footnoted.
```

15. Footnotes are processed in an environment that is different from that of the body of the text (see the .ev request in [9]).

2. Labelled footnote:

```
This is a labeled*
.FS *
The footnote is labeled with an asterisk.
.FE
footnote.
```

The text of the footnote (enclosed within the .FS-.FE pair) should *immediately* follow the word to be footnoted in the input text, so that "*F" or *label* occurs at the end of a line of input and the next line is the .FS macro call. It is also good practice to append a unpaddable space (3.3) to "*F" or *label* when they follow an end-of-sentence punctuation mark (i.e., period, question mark, exclamation point).

Appendix C illustrates the various available footnote styles as well as numbered and labeled footnotes.

8.3 Format of Footnote Text •

.FD [arg] [1]

Within the footnote text, the user can control the formatting style by specifying text hyphenation, right margin justification, and text indentation, as well as left- or right-justification of the label when text indenting is used. The .FD macro is invoked to select the appropriate style. The first argument is a number from the left column of the following table. The formatting style for each number is given by the remaining four columns. For further explanation of the first two of these columns, see the definitions of the .ad, .hy, .na, and .nh requests in [9].

0	.nh	.ad	text indent	label left justified
1	.hy	.ad	"	"
2	.nh	.na	"	"
3	.hy	.na	"	"
4	.nh	.ad	no text indent	"
5	.hy	.ad	"	"
6	.nh	.na	"	"
7	.hy	.na	"	"
8	.nh	.ad	text indent	label right justified
9	.hy	.ad	"	"
10	.nh	.na	"	"
11	.hy	.na	"	"

If the first argument to .FD is out of range, the effect is as if .FD 0 were specified. If the first argument is omitted or null, the effect is equivalent to .FD 10 in *nroff* and to .FD 0 in *troff*; these are also the respective initial defaults.

If a second argument is specified, then whenever a first-level heading is encountered, automatically-numbered footnotes begin again with 1. This is most useful with the "section-page" page numbering scheme. As an example, the input line:

.FD "" 1

maintains the default formatting style and causes footnotes to be numbered afresh after each first-level heading.

For long footnotes that continue onto the following page, it is possible that, if hyphenation is permitted, the last line of the footnote on the current page will be hyphenated. Except for this case (over which the user has control by specifying an *even* argument to .FD), hyphenation across pages is inhibited by PWB/MM.

Footnotes are separated from the body of the text by a short rule. Footnotes that continue to the next page are separated from the body of the text by a full-width rule. In *troff*, footnotes are set in type that

is two points smaller than the point size used in the body of the text.

8.4 Spacing between Footnote Entries

Normally, one blank line (a three-point vertical space) separates the footnotes when more than one occurs on a page. To change this spacing, set the register *Fs* to the desired value. For example:

```
.nr Fs 2
```

will cause two blank lines (a six-point vertical space) to occur between footnotes.

9. PAGE HEADERS AND FOOTERS

Text that occurs at the top of each page is known as the *page header*. Text printed at the bottom of each page is called the *page footer*. There can be up to three lines of text associated with the header: every page, even page only, and odd page only. Thus the page header may have up to two lines of text: the line that occurs at the top of every page and the line for the even- or odd-numbered page. The same is true for the page footer.

This section first describes the default appearance of page headers and page footers, and then the ways of changing them. We use the term *header (not* qualified by *even* or *odd)* to mean the line of the page header that occurs on every page, and similarly for the term *footer*.

9.1 Default Headers and Footers

By default, each page has a centered page number as the header {9.2}. There is no default footer and no even/odd default headers or footers, except as specified in {9.9}.

In a memorandum or a released paper, the page header on the first page is automatically suppressed *provided* a break does *not* occur before .MT is called. The macros and text of {6.9} and of {9} as well as .nr and .ds requests do *not* cause a break and are permitted before the .MT macro call.

9.2 Page Header

```
.PH [arg]
```

For this and for the .EH, .OH, .PF, .EF, .OF macros, the argument is of the form:

```
"'left-part'center-part'right-part'"
```

If it is inconvenient to use the apostrophe (') as the delimiter (i.e., because it occurs within one of the parts), it may be replaced *uniformly* by *any* other character. On output, the parts are left-justified, centered, and right-justified, respectively. See {9.11} for examples.

The .PH macro specifies the header that is to appear at the top of every page. The initial value (as stated in {9.1}) is the default centered page number enclosed by hyphens. See the top of this page for an example of this default header.

If *debug mode* is set using the flag -rD1 on the command line {2.4}, additional information, printed at the top left of each page, is included in the default header. This consists of the SCCS [10] Release and Level of PWB/MM (thus identifying the current version {11.3}), followed by the current line number within the current input file.

9.3 Even-Page Header

```
.EH [arg]
```

The .EH macro supplies a line to be printed at the top of each even-numbered page, immediately *following* the header. The initial value is a blank line.

9.4 Odd-Page Header

```
.OH [arg]
```

This macro is the same as .EH, except that it applies to odd-numbered pages.

9.5 Page Footer

.PF [arg]

The .PF macro specifies the line that is to appear at the bottom of each page. Its initial value is a blank line. If the -rC*n* flag is specified on the command line (2.4), the type of copy *follows* the footer on a separate line. In particular, if -rC3 (DRAFT) is specified, then, in addition, the footer is initialized to contain the date (6.7.1), instead of being a blank line.

9.6 Even-Page Footer

.EF [arg]

The .EF macro supplies a line to be printed at the bottom of each even-numbered page, immediately *preceding* the footer. The initial value is a blank line.

9.7 Odd-Page Footer

.OF [arg]

This macro is the same as .EF, except that it applies to odd-numbered pages.

9.8 Footer on the First Page

By default, the footer is a blank line. If, in the input text, one specifies .PF and/or .OF before the end of the first page of the document, then these lines will appear at the bottom of the first page.

The header (whatever its contents) *replaces* the footer *on the first page only* if the -rN1 flag is specified on the command line (2.4).

9.9 Default Header and Footer with "Section-Page" Numbering

Pages can be numbered sequentially within sections (4.5). To obtain this numbering style, specify -rN3 on the command line. In this case, the default *footer* is a centered "section-page" number, e.g. 3-5, and the default page header is blank.

9.10 Use of Strings and Registers in Header and Footer Macros •

String and register names may be placed in the arguments to the header and footer macros. If the value of the string or register is to be computed *when the respective header or footer is printed*, the invocation must be escaped by four (4) backslashes. This is because the string or register invocation will be processed three times:

- as the argument to the header or footer macro;
- in a formatting request within the header or footer macro;
- in a .tl request during header or footer processing.

For example, the page number register *P* must be escaped with four backslashes in order to specify a header in which the page number is to be printed at the right margin, e.g.:

```
.PH ""'Page \\\\nP'"
```

creates a right-justified header containing the word "Page" followed by the page number. Similarly, to specify a footer with the "section-page" style, one specifies (see (4.2.2.5) for meaning of *H1*):

```
.PF "'- \\\\n(H1-\\\\nP -'"
```

As another example, suppose that the user arranges for the string *a1* to contain the current section heading which is to be printed at the bottom of each page. The .PF macro call would then be:

```
.PF "'\\\\*(a1'"
```

If only one or two backslashes were used, the footer would print a constant value for *a1*, namely, its value when the .PF appeared in the input text.

9.11 Header and Footer Example •

The following sequence specifies blank lines for the header and footer lines, page numbers on the outside edge of each page (i.e., top left margin of even pages and top right margin of odd pages), and "Revision 3" on the top inside margin of each page:

```
.PH ""
.PF ""
.EH "'\\\\nP''Revision 3'"
.OH "'Revision 3''\\\\nP'"
```

9.12 Generalized Top-of-Page Processing •

☛ *This section is intended only for users accustomed to writing formatter macros.*

During header processing, PWB/MM invokes two user-definable macros. One, the .TP macro, is invoked in the environment (see .ev request in [9]) of the header; the other, .PX, is a user-exit macro that is invoked (without arguments) when the normal environment has been restored, and with "no-space" mode already in effect.

The effective initial definition of .TP (after the first page of a document) is:

```
.de TP
.sp
.tl \\•()t
.if e 'tl \\•()e
.if o 'tl \\•()o
.sp
..
```

The string }*t* contains the header, the string }*e* contains the even-page header, and the string }*o* contains the odd-page header, as defined by the .PH, .EH, and .OH macros, respectively. To obtain more specialized page titles, the user may redefine the .TP macro to cause any desired header processing [11.5]. Note that formatting done within the .TP macro is processed in an environment different from that of the body.

For example, to obtain a page header that includes three centered lines of data, say, a document's number, issue date, and revision date, one could define .TP as follows:

```
.de TP
.sp
.ce 3
777-888-999
Iss. 2, AUG 1977
Rev. 7, SEP 1977
.sp
..
```

The .PX macro may be used to provide text that is to appear at the top of each page after the normal header and that may have tab stops to align it with columns of text in the body of the document.

9.13 Generalized Bottom-of-Page Processing

```
.BS
zero or more lines of text
.BE
```

Lines of text that are specified between the .BS (bottom-block start) and .BE (bottom-block end) macros will be printed at the bottom of each page, after the footnotes (if any), but before the page footer. This block of text is removed by specifying an empty block, i.e.:

```
.BS
.BE
```

10. TABLE OF CONTENTS AND COVER SHEET

The table of contents and the cover sheet for a document are produced by invoking the .TC and .CS macros, respectively. The appropriate -rB*n* option (2.4) must *also* be specified on the command line. These macros should normally appear only once at the *end* of the document, after the Signature Block (6.11.1) and Notations (6.11.2) macros. They may occur in either order.

The table of contents is produced at the end of the document because the entire document must be processed before the table of contents can be generated. Similarly, the cover sheet is often not needed, and is therefore produced at the end.

10.1 Table of Contents

.TC [slevel] [spacing] [tlevel] [tab] [head1] [head2] [head3] [head4] [head5]

The .TC macro generates a table of contents containing the headings that were saved for the table of contents as determined by the value of the *Cl* register (4.4). Note that -rB1 or -rB3 (2.4) must also be specified to the formatter on the command line. The arguments to .TC control the spacing before each entry, the placement of the associated page number, and additional text on the first page of the table of contents before the word "CONTENTS."

Spacing before each entry is controlled by the first two arguments; headings whose level is less than or equal to *slevel* will have *spacing* blank lines (halves of a vertical space) before them. Both *slevel* and *spacing* default to 1. This means that first-level headings are preceded by one blank line (½ a vertical space). Note that *slevel* does *not* control what levels of heading have been saved; the saving of headings is the function of the *Cl* register (4.4).

The third and fourth arguments control the placement of the page number for each heading. The page numbers can be justified at the right margin with either blanks or dots ("leaders") separating the heading text from the page number, or the page numbers can follow the heading text. For headings whose level is less than or equal to *tlevel* (default 2), the page numbers are justified at the right margin. In this case, the value of *tab* determines the character used to separate the heading text from the page number. If *tab* is 0 (the default value), dots (i.e., leaders) are used; if *tab* is greater than 0, spaces are used. For headings whose level is greater than *tlevel*, the page numbers are separated from the heading text by two spaces (i.e., they are "ragged right").

All additional arguments (e.g., *head1*, *head2*, etc.), if any, are horizontally centered on the page, and precede the actual table of contents itself.

If the .TC macro is invoked with at most four arguments, then the user-exit macro .TX is invoked (without arguments) before the word "CONTENTS" is printed. By defining .TX and invoking .TC with at most four arguments, the user can specify what needs to be done at the top of the (first) page of the table of contents. For example, the following input:

```
.de TX
.ce 2
Special Application
Message Transmission
.sp 2
.in +10n
Approved: \l'3i'
.in
.sp
..
.TC
```

yields:

Special Application
Message Transmission

Approved: _____

CONTENTS

⋮

10.2 Cover Sheet

.CS [pages] [other] [total] [figs] [tbls] [refs]

The .CS macro generates a cover sheet in either the TM or released-paper style.[16] All of the other information for the cover sheet is obtained from the data given before the .MT macro call (6.9). If the released-paper style is used, all arguments to .CS are ignored. If a memorandum style is used, the .CS macro generates the "Cover Sheet for Technical Memorandum." The arguments provide the data that appears in the lower left corner of the TM cover sheet [2]: the number of pages of text, the number of other pages, the total number of pages, the number of figures, the number of tables, and the number of references.

11. MISCELLANEOUS FEATURES

11.1 Bold, Italic, and Roman

.B [bold-arg] [previous-font-arg]
.I [italic-arg] [previous-font-arg]
.R

When called without arguments, .B (or .I) changes the font to bold (or italic) in *troff*, and initiates underlining in *nroff*.[17] This condition continues until the occurrence of a .R, when the regular roman font is restored. Thus,

```
.I
here is some text.
.R
```

yields:

here is some text.

If .B or .I is called with one argument, that argument is printed in the appropriate font (underlined in *nroff*). Then the *previous* font is restored (underlining is turned off in *nroff*). If two arguments are given to a .B or .I, the second argument is then concatenated to the first with no intervening space, but is printed in the previous font (not underlined in *nroff*). For example:

16. But only if -rB2 or -rB3 has been specified on the command line.

17. For ease of explanation, in this section (11.1) *troff* behavior is described first, the convention of (1.2) not withstanding.

```
.I italic
text
.I right -justified
```

produces:

italic text *right*-justified

One can use both bold and italic fonts if one intends to use *troff*, but the *nroff* version of the output does not distinguish between bold and italic. It is probably a good idea to use .I only, unless bold is truly required. Note that font changes in headings are handled separately (4.2.2.4.1).

Anyone using a terminal that cannot underline might wish to insert:

```
.rm ul
.rm cu
```

at the beginning of the document to eliminate *all* underlining.

11.2 Justification of Right Margin

```
.SA [arg]
```

The .SA macro is used to set right-margin justification for the main body of text. Two justification flags are used: *current* and *default*. .SA 0 sets both flags to no justification, i.e., it acts like the .na request. .SA 1 is the inverse: it sets both flags to cause justification, just like the .ad request. However, calling .SA *without* an argument causes the *current* flag to be copied from the *default* flag, thus performing either a .na or .ad, depending on what the *default* is. Initially, both flags are set for no justification in *nroff* and for justification in *troff*.

In general, the request .na can be used to ensure that justification is turned off, but .SA should be used to restore justification, rather than the .ad request. In this way, justification or lack thereof for the remainder of the text is specified by inserting .SA 0 or .SA 1 *once* at the beginning of the document.

11.3 SCCS Release Identification

The string *RE* contains the SCCS [10] Release and Level of the current version of PWB/MM. For example, typing:

```
This is version \*(RE of the macros.
```

produces:

This is version 15.103 of the macros.

This information is useful in analyzing suspected bugs in PWB/MM. The easiest way to have this number appear in your output is to specify -rD1 (2.4) on the command line, which causes the string *RE* to be output as part of the page header (9.2).

11.4 Two-Column Output

PWB/MM can print two columns on a page:

```
.2C
text and formatting requests (except another .2C)
.1C
```

The .2C macro begins two-column processing which continues until a .1C macro is encountered. In two-column processing, each physical page is thought of as containing two columnar "pages" of equal (but smaller) "page" width. Page headers and footers are *not* affected by two-column processing. The .2C macro does *not* "balance" two-column output.

It is possible to have full-page width footnotes and displays when in two column mode, although the default action is for footnotes and displays to be narrow in two column mode and wide in one column

mode. Footnote and display width is controlled by a macro, .WC (Width Control), which takes the following arguments:

N	Normal default mode (−WF, −FF, −WD)
WF	Wide Footnotes always (even in two column mode)
−WF	DEFAULT: turn off WF (footnotes follow column mode, wide in 1C mode, narrow in 2C mode, unless FF is set)
FF	First Footnote; all footnotes have the same width as the *first* footnote encountered for that page
−FF	DEFAULT: turn off FF (footnote style follows the settings of WF or −WF)
WD	Wide Displays always (even in two column mode)
−WD	DEFAULT: Displays follow whichever column mode is in effect when the display is encountered

For example: .WC WD FF will cause all displays to be wide, and all footnotes on a page to be the same width, while .WC N will reinstate the default actions. If conflicting settings are given to .WC the last one is used. That is, .WC WF −WF has the effect of .WC −WF. Note that by default all options are turned off.

11.5 Column Headings for Two-Column Output •

☛ *This section is intended only for users accustomed to writing formatter macros.*

In two-column output, it is sometimes necessary to have headers over each column, as well as headers over the entire page [9]. This is accomplished by redefining the .TP macro [9.12] to provide header lines both for the entire page and for each of the columns. For example:

```
.de TP
.sp 2
.tl 'Page \\nP'OVERALL"
.tl "TITLE"
.sp
.nf
.ta 16C 31R 34 50C 65R
left—center—right—left—center—right                (where — stands for the tab character)
—first column———second column
.fi
.sp 2
..
```

The above example will produce two lines of page header text plus two lines of headers over each column. The tab stops are for a 65-en overall line length.

11.6 Vertical Spacing

.SP [lines]

There exist several ways of obtaining vertical spacing, all with different effects.

The .sp request spaces the number of lines specified, *unless* "no space" (.ns) mode is on, in which case the request is ignored. This mode is typically set at the end of a page header in order to eliminate spacing by a .sp or .bp request that just happens to occur at the top of a page. This mode can be turned

off via the .rs ("restore spacing") request.

The .SP macro is used to avoid the accumulation of vertical space by successive macro calls. Several .SP calls in a row produce *not* the sum of their arguments, but their maximum; i.e., the following produces only 3 blank lines:

```
.SP 2
.SP 3
.SP
```

Many PWB/MM macros utilize .SP for spacing. For example, ".LE 1" (5.3.2) immediately followed by ".P" (4.1) produces only a single blank line (½ a vertical space) between the end of the list and the following paragraph. An omitted argument defaults to one blank line (*one* vertical space). Unscaled fractional amounts are permitted; like .sp, .SP is also inhibited by the .ns request.

11.7 Skipping Pages

```
.SK [pages]
```

The .SK macro skips pages, but retains the usual header and footer processing. If *pages* is omitted, null, or 0, .SK skips to the top of the next page *unless* it is currently at the top of a page, in which case it does nothing. .SK *n* skips *n* pages. That is, .SK always positions the text that follows it at the top of a page, while .SK 1 always leaves one page that is blank except for the header and footer.

11.8 FORCING AN ODD PAGE

```
.OP
```

This macro is used to ensure that the following text begins at the top of an odd-numbered page. If currently at the top of an odd page, no motion takes place. If currently on an even page, text resumes printing at the top of the next page. If currently on an odd page (but not at the top of the page) one blank page is produced, and printing resumes on the page after that.

11.9 Setting Point Size and Vertical Spacing

In *troff*, the default point size (obtained from the register *S* (2.4)) is 10, with a vertical spacing of 12 points (i.e., 6 lines per inch). The prevailing point size and vertical spacing may be changed by invoking the .S macro:

```
.S [arg]
```

If *arg* is null, the *previous* point size is restored. If *arg* is negative, the point size is decremented by the specified amount. If *arg* is *signed* positive, the point size is incremented by the specified amount, and if *arg* is unsigned, it is used as the new point size; if *arg* is greater than 99, the *default* point size (10) is restored. Vertical spacing is always two points greater than the point size.[18]

12. ERRORS AND DEBUGGING

12.1 Error Terminations

When a macro discovers an error, the following actions occur:

- A break occurs.

---

18. Footnotes (8) are printed in a size two points *smaller* than the point size of the body, with an additional vertical spacing of three points between footnotes.

- To avoid confusion regarding the location of the error, the formatter output buffer (which may contain some text) is printed.
- A short message is printed giving the name of the macro that found the error, the type of error, and the approximate line number (in the current input file) of the last processed input line. (All the error messages are explained in Appendix E.)
- Processing terminates, unless the register *D* {2.4} has a positive value. In the latter case, processing continues even though the output is guaranteed to be deranged from that point on.

☞ *The error message is printed by writing it directly to the user's terminal. If an output filter, such as* 300(1), 450(1), *or* hp(1) *is being used to post-process* nroff *output, the message may be garbled by being intermixed with text held in that filter's output buffer.*

☞ *If either* tbl(1) *or* eqn(1)/neqn(1), *or both are being used, and if the* -olist *option of the formatter causes the last page of the document* not *to be printed, a harmless "broken pipe" message results.*

12.2 Disappearance of Output

This usually occurs because of an unclosed diversion (e.g., missing .FE or .DE). Fortunately, the macros that use diversions are careful about it, and they check to make sure that illegal nestings do not occur. If any message is issued about a missing .DE or .FE, the appropriate action is to search backwards from the termination point looking for the corresponding .DS, .DF, or .FS.

The following command:

```
grep -n "^\.[EDFT][EFNQS]" files ...
```

prints all the .DS, .DF, .DE, .FS, .FE, .TS, .TE, .EQ, and .EN macros found in *files* ..., each preceded by its file name and the line number in that file. This listing can be used to check for illegal nesting and/or omission of these macros.

13. EXTENDING AND MODIFYING THE MACROS •

13.1 Naming Conventions

In this section, the following conventions are used to describe legal names:

- n: digit
- a: lower-case letter
- A: upper-case letter
- x: any letter or digit (any alphanumeric character)
- s: special character (any non-alphanumeric character)

All other characters are literals (i.e., stand for themselves).

Note that *request*, *macro*, and *string* names are kept by the formatters in a single internal table, so that there must be no duplication among such names. *Number register* names are kept in a separate table.

13.1.1 Names Used by Formatters.

requests:
- aa (most common)
- an (only one, currently: .c2)

registers:
- aa (normal)
- .x (normal)
- .s (only one, currently: .S)
- % (page number)

13.1.2 Names Used by PWBMM.

macros:
- AA (most common, accessible to user)
- A (less common, accessible to user)

)x (internal, constant)
>x (internal, dynamic)

strings: AA (most common, accessible to user)
A (less common, accessible to user)
]x (internal, usually allocated to specific functions throughout)
}x (internal, more dynamic usage)

registers: Aa (most common, accessible to users)
An (common, accessible to user)
A (accessible, set on command line)
:x (mostly internal, rarely accessible, usually dedicated)
;x (internal, dynamic, temporaries)

13.1.3 Names Used by EQN/NEQN and TBL. The equation preprocessors, *eqn*(1) and *neqn*(1), use registers *and* string names of the form *nn*. The table preprocessor, *tbl*(1), uses names of the form:

2- 2+ a| nn #a ## #- #^ ^a T& TW

13.1.4 User-Definable Names. After the above, what is left for user extensions? To avoid problems, we suggest using names that consist either of a single lower-case letter, or of a lower-case letter followed by anything other than a lower-case letter. The following is a sample naming convention:

macros: aA
Aa

strings: a
a) (or a|, or a}, etc.)

registers a
aA

13.2 Sample Extensions

13.2.1 Appendix Headings. The following gives a way of generating and numbering appendices:

```
.nr Hu 1
.nr a 0
.de aH
.nr a +1
.nr P 0
.PH "'''Appendix \\na - \\\\\\\\nP'"
.SK
.HU "\\$1"
..
```

After the above initialization and definition, each call of the form ".aH "title"" begins a new page (with the page header changed to "Appendix *a* - *n*") and generates an unnumbered heading of *title*, which, if desired, can be saved for the table of contents. Those who wish Appendix titles to be centered must, in addition, set the register *Hc* to 1 {4.2.2.3}.

13.2.2 Hanging Indent with Tabs. The following example illustrates the use of the hanging-indent feature of variable-item lists {5.3.3.6}. First, a user-defined macro is built to accept four arguments that make up the *mark*. Each argument is to be separated from the previous one by a tab character; tab settings are defined later. Since the first argument may begin with a period or apostrophe, the "\&" is used so that the formatter will not interpret such a line as a formatter request or macro.[19] The "\t" is

translated by the formatter into a tab character. The "\c" is used to concatenate the line of *text* that follows the macro to the line of text built by the macro. The macro definition and an example of its use are as follows:

```
.de aX
.LI
\&\\$1\t\\$2\t\\$3\t\\$4\t\c
..
  :
  :
.ta 9n 18n 27n 36n
.VL 36
.aX .nh off \- no
No hyphenation.
Automatic hyphenation is turned off.
Words containing hyphens
(e.g., mother-in-law) may still be split across lines.
.aX .hy on \- no
Hyphenate.
Automatic hyphenation is turned on.
.aX .hc\□c none none no                    (□ stands for a space)
Hyphenation indicator character is set to "c" or removed.
During text processing the indicator is suppressed
and will not appear in the output.
Prepending the indicator to a word has the effect
of preventing hyphenation of that word.
.LE
```

The resulting output is:

.nh	off	–	no	No hyphenation. Automatic hyphenation is turned off. Words containing hyphens (e.g., mother-in-law) may still be split across lines.
.hy	on	–	no	Hyphenate. Automatic hyphenation is turned on.
.hc c	none	none	no	Hyphenation indicator character is set to "c" or removed. During text processing the indicator is suppressed and will not appear in the output. Prepending the indicator to a word has the effect of preventing hyphenation of that word.

14. CONCLUSION

The following are the qualities that we have tried to emphasize in PWB/MM, in approximate order of importance:

- *Robustness in the face of error*—A user need not be an *nroff/troff* expert to use these macros. When the input is incorrect, either the macros attempt to make a reasonable interpretation of the error, or a message describing the error is produced. We have tried to minimize the possibility that a user would get cryptic system messages or strange output as a result of simple errors.

---

19. The two-character sequence "\&" is understood by the formatters to be a "zero-width" space, i.e., it causes no output characters to appear.

- *Ease of use for simple documents*—It is not necessary to write complex sequences of commands to produce simple documents. Reasonable default values are provided, where at all possible.
- *Parameterization*—There are many different preferences in the area of document styling. Many parameters are provided so that users can adapt the output to their respective needs over a wide range of styles.
- *Extension by moderately expert users*—We have made a strong effort to use mnemonic naming conventions and consistent techniques in the construction of the macros. Naming conventions are given so that a user can add new macros or redefine existing ones, if necessary.
- *Device independence*—The most common use of PWB/MM is to print documents on hard-copy typewriter terminals, using the *nroff* formatter. The macros can be used conveniently with both 10- and 12-pitch terminals. In addition, output can be scanned with an appropriate CRT terminal. The macros have been constructed to allow compatibility with *troff*, so that output can be produced both on typewriter-like terminals and on a phototypesetter.
- *Minimization of input*—The design of the macros attempts to minimize repetitive typing. For example, if a user wants to have a blank line after all first- or second-level headings, he or she need only set a specific parameter *once* at the beginning of a document, rather than add a blank line after each such heading.
- *Decoupling of input format from output style*—There is but one way to prepare the input text, although the user may obtain a number of output styles by setting a few global flags. For example, the .H macro is used for all numbered headings, yet the actual output style of these headings may be made to vary from document to document or, for that matter, within a single document.

Future releases of PWB/MM will provide additional features that are found to be useful. The authors welcome comments, suggestions, and criticisms of the macros and of this manual.

Acknowledgements. We are indebted to T. A. Dolotta for his continuing guidance during the development of PWB/MM. We also thank our many users who have provided much valuable feedback, both about the macros and about this manual. Many of the features of PWB/MM are patterned after similar features in a number of earlier macro packages, and, in particular, after one implemented by M. E. Lesk. Finally, because PWB/MM often approaches the limits of what is possible with the text formatters, during the implementation of PWB/MM we have generated atypical requirements and encountered unusual problems; we thank J. F. Ossanna for his willingness to add new features to the formatters and to invent ways of having the formatters perform unusual but desired actions.

References

[1] Dolotta, T. A., Haight, R. C., and Piskorik, E. M., eds. *PWB/UNIX User's Manual—Edition 1.0.* Bell Laboratories, May 1977.

[2] Bell Laboratories, Methods and Systems Department. Office Guide. Unpublished Memorandum, Bell Laboratories, April 1972 (as revised).

[3] Kernighan, B. W. UNIX for Beginners. Bell Laboratories, October 1974.

[4] Kernighan, B. W. A Tutorial Introduction to the UNIX Text Editor. Bell Laboratories, October 1974.

[5] Kernighan, B. W. A TROFF Tutorial. Bell Laboratories, August 1976.

[6] Kernighan, B. W., and Cherry, L. L. Typesetting Mathematics—User's Guide (Second Edition). Bell Laboratories, June 1976.

[7] Scrocca, C. New Graphic Symbols for EQN and NEQN. Bell Laboratories, September 1976.

[8] Smith, D. W., and Piskorik, E. M. Typing Documents with PWB/MM. Bell Laboratories, October 1977.

[9] Ossanna, J. F. NROFF/TROFF User's Manual. Bell Laboratories, October 1976.

[10] Bonanni, L. E., and Glasser, A. L. SCCS/PWB User's Manual. Bell Laboratories, November 1977.

[11] Lesk, M. Tbl—A Program to Format Tables. Bell Laboratories, September 1977.

Appendix A: DEFINITIONS OF LIST MACROS •

☛ *This appendix is intended only for users accustomed to writing formatter macros.*

Here are the definitions of the list-initialization macros {5.3.3}:[20]

```
.de AL
.if!e\\S1ee .if!e\\S1e1e .if!e\\S1eae .if!e\\S1eAe .if!e\\S1ele .if!e\\S1eie .)D"AL:bad arg:\\S1
.if \\n(.S<3 \{.ie \we\\S2e=0 .)L \\n(Lin 0 \\n(Lin-\we\0\0.eu 1 "\\S1"
.el .LB 0\\S2 0 2 1 "\\S1" \}
.if \\n(.S>2 \{.ie \we\\S2e=0 .)L \\n(Lin 0 \\n(Lin-\we\0\0.eu 1 "\\S1" 0 1
.el .LB 0\\S2 0 2 1 "\\S1" 0 1 \}
..
.de BL
.nr ;0 \\n(Pi
.if \\n(.S>0 .if \we\\S1e>0 .nr ;0 0\\S1
.if \\n(.S<2 .LB \\n(;0 0 1 0 \\•(BU
.if \\n(.S>1 .LB \\n(;0 0 1 0 \\•(BU 0 1
.rr ;0
..
.de DL
.nr ;0 \\n(Pi
.if \\n(.S>0 .if \we\\S1e>0 .nr ;0 0\\S1
.if \\n(.S<2 .LB \\n(;0 0 1 0 \(em
.if \\n(.S>1 .LB \\n(;0 0 1 0 \(em 0 1
.rr ;0
..
.de ML
.if !\\n(.$ .)D "ML:missing arg"
.nr ;0 \we\\S1eu/3u/\\n(.su+1u\" get size in n's
.if !\\n(.$-1 .LB \\n(;0 0 1 0 "\\S1"
.if \\n(.$-1 .if !\\n(.$-2 .LB 0\\S2 0 1 0 "\\S1"
.if \\n(.$-2 .if !\we\\S2e .LB \\n(;0 0 1 0 "\\S1" 0 1
.if \\n(.$-2 .if \we\\S2e .LB 0\\S2 0 1 0 "\\S1" 0 1
..
.de RL
.nr ;0 6
.if \\n(.S>0 .if \we\\S1e>0 .nr ;0 0\\S1
.if \\n(.S<2 .LB \\n(;0 0 2 4
.if \\n(.S>1 .LB \\n(;0 0 2 4 1 0 1
.rr ;0
..
.de VL
.if !\\n(.$ .)D "VL:missing arg"
.if !\\n(.$-2 .LB 0\\S1 0\\S2 0 0
.if \\n(.$-2 .LB 0\\S1 0\\S2 0 0 \& 0 1
..
```

20. On this page, e represents the BEL character, .)D is an internal PWB/MM macro that prints error messages, and .)L is similar to .LB, except that it expects its arguments to be scaled.

Any of these can be redefined to produce different behavior: e.g., to provide two spaces between the bullet of a bullet item and its text, redefine .BL as follows before invoking it:[21]

```
.de BL
.LB 3 0 2 0 \\•(BU
..
```

21. With this redefinition, .BL cannot have any arguments.

Appendix B: USER-DEFINED LIST STRUCTURES •

☛ *This appendix is intended only for users accustomed to writing formatter macros.*

If a large document requires complex list structures, it is useful to be able to define the appearance for each list level only once, instead of having to define it at the beginning of each list. This permits consistency of style in a large document. For example, a generalized list-initialization macro might be defined in such a way that what it does depends on the list-nesting level list nesting in effect at the time the macro is called. Suppose that levels 1 through 5 of lists are to have the following appearance:

A.

[1]

•

a)

+

The following code defines a macro (.aL) that always begins a new list and determines the type of list according to the current list level. To understand it, you should know that the number register .g is used by the PWB/MM list macros to determine the current list level; it is 0 if there is no currently active list. Each call to a list-initialization macro increments .g, and each .LE call decrements it.

```
.de aL
'\"       register g is used as a local temporary to save :g before it is changed below
.nr g \\n(:g
.if \\ng=0 .AL A \" give me an A.
.if \\ng=1 .LB \\n(Li 0 1 4 \" give me a [1]
.if \\ng=2 .BL \" give me a bullet
.if \\ng=3 .LB \\n(Li 0 2 2 a \" give me an a)
.if \\ng=4 .ML + \" give me a +
..
```

This macro can be used (in conjunction with .LI and .LE) instead of .AL, .RL, .BL, .LB, and .ML. For example, the following input:

```
.aL
.LI
first line.
.aL
.LI
second line.
.LE
.LI
third line.
.LE
```

will yield:

A. first line.

[1] second line.

B. third line.

There is another approach to lists that is similar to the .H mechanism. The list-initialization, as well as the .LI and the .LE macros are all included in a single macro. That macro (called .bL below) requires

an argument to tell it what level of item is required; it adjusts the list level by either beginning a new list or setting the list level back to a previous value, and then issues a .LI macro call to produce the item:

```
.de bL
.ie \\n(.$ .nr g \\$1 \" if there is an argument, that is the level
.el .nr g \\n(:g \" if no argument, use current level
.if \\ng-\\n(:g>1 .)D "-ILLEGAL SKIPPING OF LEVEL" \" increasing level by more than 1
.if \\ng>\\n(:g \{.aL \\ng-1 \" if g > :g, begin new list.
.       nr g \\n(:g\} \" and reset g to current level (.aL changes g)
.if \\n(:g>\\ng .LC \\ng \" if :g > g, prune back to correct level
'\"       if :g = g, stay within current list
.LI \" in all cases, get out an item
..
```

For .bL to work, the previous definition of the .aL macro must be changed to obtain the value of *g* from its argument, rather than from .g. Invoking .bL without arguments causes it to stay at the current list level. The PWB/MM .LC macro (List Clear) removes list descriptions until the level is less than or equal to that of its argument. For example, the .H macro includes the call ".LC 0". If text is to be resumed at the end of a list, insert the call ".LC 0" to clear out the lists completely. The example below illustrates the relatively small amount of input needed by this approach. The input text:

```
The quick brown fox jumped over the lazy dog's back.
.bL 1
first line.
.bL 2
second line.
.bL 1
third line.
.bL
fourth line.
.LC 0
fifth line.
```

yields:

The quick brown fox jumped over the lazy dog's back.

A. first line.

   [1] second line.

B. third line.

C. fourth line.

fifth line.

Appendix C: SAMPLE FOOTNOTES

The following example illustrates several footnote styles and both labeled and automatically-numbered footnotes. The actual input for the immediately following text and for the footnotes at the bottom of this page is shown on the following page:

With the footnote style set to the *nroff* default, we process a footnote[1] followed by another one.[*****] Using the .FD macro, we changed the footnote style to hyphenate, right margin justification, indent, and left justify the label. Here is a footnote,[2] and another.[†] The footnote style is now set, again via the .FD macro, to no hyphenation, no right margin justification, no indentation, and with the label left-justified. Here comes the final one.[3]

1. This is the first footnote text example (.FD 10). This is the default style for *nroff*. The right margin is *not* justified. Hyphenation is *not* permitted. The text is indented, and the automatically generated label is *right*-justified in the text-indent space.

[*****] This is the second footnote text example (.FD 10). This is also the default *nroff* style but with a long footnote label provided by the user.

2. This is the third footnote example (.FD 1). The right margin is justified, the footnote text is indented, the label is *left*-justified in the text-indent space. Although not necessarily illustrated by this example, hyphenation is permitted. The quick brown fox jumped over the lazy dog's back.

[†] This is the fourth footnote example (.FD 1). The style is the same as the third footnote.

3. This is the fifth footnote example (.FD 6). The right margin is *not* justified, hyphenation is *not* permitted, the footnote text is *not* indented, and the label is placed at the beginning of the first line. The quick brown fox jumped over the lazy dog's back. Now is the time for all good men to come to the aid of their country.

```
.FD 10
With the footnote style set to the
.I nroff
default, we process a footnote\•F
.FS
This is the first footnote text example (.FD 10).
This is the default style for
.I nroff.
The right margin is
.I not
justified.
Hyphenation is
.I not
permitted.
The text is indented, and the automatically generated label is
.I right -justified
in the text-indent space.
.FE
followed by another one.•••••\□                    (□ stands for a space)
.FS •••••
This is the second footnote text example (.FD 10).
This is also the default
.I nroff
style but with a long footnote label provided by the user.
.FE
.FD 1
Using the .FD macro, we changed the footnote style to hyphenate, right margin justification,
indent, and left justify the label.
Here is a footnote.\•F
.FS
This is the third footnote example (.FD 1).
The right margin is justified, the footnote text is indented, the label is
.I left -justified
in the text-indent space.
Although not necessarily illustrated by this example, hyphenation is permitted.
The quick brown fox jumped over the lazy dog's back.
.FE
and another.\(dg\□
.FS \(dg
This is the fourth footnote example (.FD 1).
The style is the same as the third footnote.
.FE
.FD 6
The footnote style is now set, again via the .FD macro, to no hyphenation, no right margin justification,
no indentation, and with the label left-justified.
Here comes the final one.\•F\□
.FS
This is the fifth footnote example (.FD 6).
The right margin is
.I not
justified, hyphenation is
.I not
permitted, the footnote text is
.I not
indented, and the label is placed at the beginning of the first line.
The quick brown fox jumped over the lazy dog's back.
Now is the time for all good men to come to the aid of their country.
.FE
```

Appendix D: SAMPLE LETTER

☛ *The* nroff *and* troff *outputs corresponding to the input text below are shown on the following pages.*

```
.ND "May 31, 1979"
.TL 334455
Out-of-Hours Course Description
.AU "D. W. Stevenson" DWS PY 9876 5432 1X-123
.MT 0
.DS
J. M. Jones:
.DE
.P
Please use the following description for the Out-of-Hours course
"Document Preparation on the PWB/UNIX"
.FS "
UNIX is a Trademark of Bell Laboratories.
.FE
time-sharing system":
.P
The course is intended for clerks, typists, and others
who intend to use the PWB/UNIX system
for preparing documentation.
The course will cover such topics as:
.VL 18
.LI Environment:
utilizing a time-sharing computer system;
accessing the system;
using appropriate output terminals.
.LI Files:
how text is stored on the system;
directories;
manipulating files.
.LI "Text editing:"
how to enter text so that subsequent revisions are easier to make;
how to use the editing system to
add, delete, and move lines of text;
how to make corrections.
.LI "Text processing:"
basic concepts;
use of general-purpose formatting packages.
.LI "Other facilities:"
additional capabilities useful to the typist such as the
.I "typo, spell, diff,"
and
.I grep
commands and a desk-calculator package.
.LE
.SG jrm
.NS
S. P. Lename
H. O. Del
M. Hill
.NE
```

Bell Laboratories

subject: Out-of-Hours Course Description
Case: 334455

date: May 31, 1979

from: D. W. Stevenson
PY 9876
IX-123 x5432

J. M. Jones:

Please use the following description for the Out-of-Hours course "Document Preparation on the PWB/UNIX* time-sharing system":

The course is intended for clerks, typists, and others who intend to use the PWB/UNIX system for preparing documentation. The course will cover such topics as:

Environment: utilizing a time-sharing computer system; accessing the system; using appropriate output terminals.

Files: how text is stored on the system; directories; manipulating files.

Text editing: how to enter text so that subsequent revisions are easier to make; how to use the editing system to add, delete, and move lines of text; how to make corrections.

Text processing: basic concepts; use of general-purpose formatting packages.

Other facilities: additional capabilities useful to the typist such as the _typo_, _spell_, _diff_, and _grep_ commands and a desk-calculator package.

PY-9876-DWS-jrm

D. W. Stevenson

Copy to
S. P. Lename
H. O. Del
M. Hill

* UNIX is a Trademark of Bell Laboratories.

Bell Laboratories

subject: Out-of-Hours Course Description
Case: 334455

date: May 31, 1979

from: D. W. Stevenson
PY 9876
1X-123 x5432

J. M. Jones:

Please use the following description for the Out-of-Hours course "Document Preparation on the PWB/UNIX* time-sharing system":

The course is intended for clerks, typists, and others who intend to use the PWB/UNIX system for preparing documentation. The course will cover such topics as:

Environment:	utilizing a time-sharing computer system; accessing the system; using appropriate output terminals.
Files:	how text is stored on the system; directories; manipulating files.
Text editing:	how to enter text so that subsequent revisions are easier to make; how to use the editing system to add, delete, and move lines of text; how to make corrections.
Text processing:	basic concepts; use of general-purpose formatting packages.
Other facilities:	additional capabilities useful to the typist such as the *typo*, *spell*, *diff*, and *grep* commands and a desk-calculator package.

PY-9876-DWS-jrm

D. W. Stevenson

Copy to
S. P. Lename
H. O. Del
M. Hill

* UNIX is a Trademark of Bell Laboratories.

Appendix E: ERROR MESSAGES

1. PWB/MM Error Messages

Each PWB/MM error message consists of a standard part followed by a variable part. The standard part is of the form:

ERROR:input line *n*:

The variable part consists of a descriptive message, usually beginning with a macro name. The variable parts are listed below in alphabetical order by macro name, each with a more complete explanation:[22]

Check TL, AU, AS, AE, MT sequence	The proper sequence of macros for the beginning of a memorandum is shown in {6.9}. Something has disturbed this order.
AL:bad arg:*value*	The argument to the .AL macro is not one of 1, A, a, I, or i. The incorrect argument is shown as *value*.
CS:cover sheet too long	The text of the cover sheet is too long to fit on one page. The abstract should be reduced or the indent of the abstract should decreased {6.4}.
DS:too many displays	More than 26 floating displays are active at once, i.e., have been accumulated but not yet output.
DS:missing FE	A display starts inside a footnote. The likely cause is the omission (or misspelling) of a .FE to end a previous footnote.
DS:missing DE	.DS or .DF occurs within a display, i.e., a .DE has been omitted or mistyped.
DE:no DS or DF active	.DE has been encountered but there has not been a previous .DS or .DF to match it.
FE:no FS	.FE has been encountered with no previous .FS to match it.
FS:missing FE	A previous .FS was not matched by a closing .FE, i.e., an attempt is being made to begin a footnote inside another one.
FS:missing DE	A footnote starts inside a display, i.e., a .DS or .DF occurs without a matching .DE.
H:bad arg:*value*	The first argument to .H must be a single digit from 1 to 7, but *value* has been supplied instead.
H:missing FE	A heading macro (.H or .HU) occurs inside a footnote.
H:missing DE	A heading macro (.H or .HU) occurs inside a display.
H:missing arg	.H needs at least 1 argument.
HU:missing arg	.HU needs 1 argument.
LB:missing arg(s)	.LB requires at least 4 arguments.
LB:too many nested lists	Another list was started when there were already 6 active lists.
LE:mismatched	.LE has occurred without a previous .LB or other list-initialization macro {5.3.3}. Although this is not a fatal error,

22. This list is set up by ".LB 37 0 2 0" {5.4}.

	the message is issued because there almost certainly exists some problem in the preceding text.
LI:no lists active	.LI occurs without a preceding list-initialization macro. The latter has probably been omitted, or has been separated from the .LI by an intervening .H or .HU.
ML:missing arg	.ML requires at least 1 argument.
ND:missing arg	.ND requires 1 argument.
SA:bad arg:value	The argument to .SA (if any) must be either 0 or 1. The incorrect argument is shown as *value*.
SG:missing DE	.SG occurs inside a display.
SG:missing FE	.SG occurs inside a footnote.
SG:no authors	.SG occurs without any previous .AU macro(s).
VL:missing arg	.VL requires at least 1 argument.

II. Formatter Error Messages

Most messages issued by the formatter are self-explanatory. Those error messages over which the *user* has (some) control are listed below. Any other error messages should be reported to the local system-support group.

"Cannot do ev" is caused by *(a)* setting a page width that is negative or extremely short, *(b)* setting a page length that is negative or extremely short, *(c)* reprocessing a macro package (e.g. performing a .so to a macro package that was requested from the command line), and *(d)* requesting the −s1 option to *troff* on a document that is longer than ten pages.

"Cannot open *filename*" is issued if one of the files in the list of files to be processed cannot be opened.

"Exception word list full" indicates that too many words have been specified in the hyphenation exception list (via .hw requests).

"Line overflow" means that the output line being generated was too long for the formatter's line buffer. The excess was discarded. See the "Word overflow" message below.

"Non-existent font type" means that a request has been made to mount an unknown font.

"Non-existent macro file" means that the requested macro package does not exist.

"Non-existent terminal type" means that the terminal options refers to an unknown terminal type.

"Out of temp file space" means that additional temporary space for macro definitions, diversions, etc. cannot be allocated. This message often occurs because of unclosed diversions (missing .FE or .DE), unclosed macro definitions (e.g., missing ".."), or a huge table of contents.

"Too many page numbers" is issued when the list of pages specified to the formatter *-o* option is too long.

"Too many string/macro names" is issued when the pool of string and macro names is full. Unneeded strings and macros can be deleted using the .rm request.

"Too many number registers" means that the pool of number register names is full. Unneeded registers can be deleted by using the .rr request.

"Word overflow" means that a word being generated exceeded the formatter's word buffer. The excess characters were discarded. A likely cause for this and for the "Line overflow" message above are very long lines or words generated through the misuse of \c or of the .cu request, or very long equations produced by *eqn*(1)/*neqn*(1).

Appendix F: SUMMARY OF MACROS, STRINGS, AND NUMBER REGISTERS

1. Macros

The following is an alphabetical list of macro names used by PWB/MM. The first line of each item gives the name of the macro, a brief description, and a reference to the section in which the macro is described. The second line gives a prototype call of the macro.

Macros marked with an asterisk are *not*, in general, invoked directly by the user. Rather, they are "user exits" called from inside header, footer, or other macros.

1C — One-column processing (11.4)
.1C

2C — Two-column processing (11.4)
.2C

AE — Abstract end (6.4)
.AE

AF — Alternate format of "Subject/Date/From" block (6.7.2)
.AF [company-name]

AL — Automatically-incremented list start (5.3.3.1)
.AL [type] [text-indent] [1]

AS — Abstract start (6.4)
.AS [arg] [indent]

AU — Author information (6.2)
.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]

B — Bold (underline in *nroff*) (11.1)
.B [bold-arg] [previous-font-arg]

BE — Bottom End (9.13)
.BE

BL — Bullet list start (5.3.3.2)
.BL [text-indent] [1]

BS — Bottom Start (9.13)
.BS

CS — Cover sheet (10.2)
.CS [pages] [other] [total] [figs] [tbls] [refs]

DE — Display end (7.1)
.DE

DF — Display floating start (7.2)
.DF [format] [fill]

DL — Dash list start (5.3.3.3)
.DL [text-indent] [1]

DS — Display static start (7.1)
.DS [format] [fill]

EC — Equation caption (7.5)
.EC [title] [override] [flag]

EF — Even-page footer (9.6)
.EF [arg]

EH Even-page header (9.3)
.EH [arg]

EN End equation display (7.4)
.EN

EQ Equation display start (7.4)
.EQ [label]

EX Exhibit caption (7.5)
.EX [title] [override] [flag]

FD Footnote default format (8.3)
.FD [arg] [1]

FE Footnote end (8.2)
.FE

FG Figure title (7.5)
.FG [title] [override] [flag]

FS Footnote start (8.2)
.FS [label]

H Heading—numbered (4.2)
.H level [heading-text]

HC Hyphenation character (3.4)
.HC [hyphenation-indicator]

HM Heading mark style (Arabic or Roman numerals, or letters) (4.2.2.5)
.HM [arg1] ... [arg7]

HU Heading—unnumbered (4.3)
.HU heading-text

HX * Heading user exit X (before printing heading) (4.6)
.HX dlevel rlevel heading-text

HZ * Heading user exit Z (after printing heading) (4.6)
.HZ dlevel rlevel heading-text

I Italic (underline in *nroff*) (11.1)
.I [italic-arg] [previous-font-arg]

LB List begin (5.4)
.LB text-indent mark-indent pad type [mark] [LI-space] [LB-space]

LC List-status clear (Appendix B)
.LC [list-level]

LE List end (5.3.2)
.LE [1]

LI List item (5.3.1)
.LI [mark] [1]

ML Marked list start (5.3.3.4)
.ML mark [text-indent] [1]

MT Memorandum type (6.6)
.MT [type] [1]

ND New date (6.7.1)
.ND new-date

NE Notation end (6.11.2)
.NE

NS Notation start (6.11.2)
.NS [arg]

OF Odd-page footer (9.7)
.OF [arg]

OH Odd-page header (9.4)
.OH [arg]

OK Other keywords for TM cover sheet (6.5)
.OK [keyword] ...

OP Odd page (11.7)
.OP

P Paragraph (4.1)
.P [type]

PF Page footer (9.5)
.PF [arg]

PH Page header (9.2)
.PH [arg]

PX * Page-header user exit (9.12)
.PX

R Return to regular (roman) font (end underlining in *nroff*) (11.1)
.R

RL Reference list start (5.3.3.5)
.RL [text-indent] [1]

S Set *troff* point size and vertical spacing (11.8)
.S [arg]

SA Set adjustment (right-margin justification) default (11.2)
.SA [arg]

SG Signature line (6.11.1)
.SG [arg] [1]

SK Skip pages (11.7)
.SK [pages]

SP Space—vertically (11.6)
.SP [lines]

TB Table title (7.5)
.TB [title] [override] [flag]

TC Table of contents (10.1)
.TC [tlevel] [spacing] [tlevel] [tab] [head1] [head2] [head3] [head4] [head5]

TE Table end (7.3)
.TE

TH Table header {7.3}
.TH [N]

TL Title of memorandum {6.1}
.TL [charging-case] [filing-case]

TM Technical Memorandum number(s) {6.3}
.TM [number] ...

TP • Top-of-page macro {9.12}
.TP

TS Table start {7.3}
.TS [H]

TX • Table-of-contents user exit {10.1}
.TX

VL Variable-item list start {5.3.3.6}
.VL text-indent [mark-indent] [1]

WC Width Control {11.4}
.WC [format]

IL. Strings

The following is an alphabetical list of string names used by PWB/MM, giving for each a brief description, section reference, and initial (default) value(s). See {1.4} for notes on setting and referencing strings.

BU Bullet {3.7}
nroff: ●
troff: •

F Footnote numberer {8.1}
nroff: \u\\n+ (:p\d
troff: \v'-.4m'\s-3\\n+ (:p\s0\v'.4m'

DT Date (current date, unless overridden) {6.7.1}
Month day, year (e.g., January 22, 1980)

EM Em dash string, produces an em dash for both *nroff* and *troff* {3.8}.

HF Heading font list, up to seven codes for heading levels 1 through 7 {4.2.2.4.1}
3 3 2 2 2 2 2 (all underlined in *nroff*, and B B I I I I I in *troff*)

HP Heading point size list, up to seven codes for heading levels 1 through 7 {4.2.2}

RE SCCS Release and Level of PWB/MM {11.3}
Release.Level (e.g., 15.103)

Tm Trademark string places the letters "TM" one half-line above the text that it follows.

Note that if the released-paper style is used, then, in addition to the above strings, certain BTL location codes are defined as strings; these location strings are needed only until the .MT macro is called {6.8}.

III. Number Registers

This section provides an alphabetical list of register names, giving for each a brief description, section reference, initial (default) value, and the legal range of values (where [m:n] means values from m to n inclusive).

Any register having a single-character name can be set from the command line. An asterisk attached to a register name indicates that that register can be set *only* from the command line or *before* the PWB/MM

macro definitions are read by the formatter (2.4, 2.5). See (1.4) for notes on setting and referencing registers.

A * Has the effect of invoking the .AF macro without an argument (2.4)
0, [0:1]

Au Inhibits printing of author's location, department, room, and extension in the "from" portion of a memorandum (6.2)
1, [0:1]

B * Defines table-of-contents and/or cover-sheet macros (2.4)
0, [0:3]

C * Copy type (Original, DRAFT, etc.) (2.4)
0 (Original), [0:3]

Cl Contents level (i.e., level of headings saved for table of contents) (4.4)
2, [0:7]

D * Debug flag (2.4)
0, [0:1]

De Display eject register for floating dislays (7.2)
0, [0:1]

Df Display format register for floating displays (7.2)
5, [0:5]

Ds Static display pre- and post-space (7.1)
1, [0:1]

Ec Equation counter, used by .EC macro (7.5)
0, [0:?], incremented by 1 for each .EC call.

Ej Page-ejection flag for headings (4.2.2.1)
0 (no eject), [0:7]

Fg Figure counter, used by .FG macro (7.5)
0, [0:?], incremented by 1 for each .FG call.

Fs Footnote space (i.e., spacing between footnotes) (3.4)
1, [0:?]

H1-H7 Heading counters for levels 1-7 (4.2.2.5)
0, [0:?], incremented by .H of corresponding level or .HU if at level given by register *Hu*. H2-H7 are reset to 0 by any heading at a lower-numbered level.

Hb Heading break level (after .H and .HU) (4.2.2.2)
2, [0:7]

Hc Heading centering level for .H and .HU (4.2.2.3)
0 (no centered headings), [0:7]

Hi Heading temporary indent (after .H and .HU) (4.2.2.2)
1 (indent as paragraph), [0:2]

Hs Heading space level (after .H and .HU) (4.2.2.2)
2 (space only after .H 1 and .H 2), [0:7]

Ht Heading type (for .H: single or concatenated numbers) (4.2.2.5)
0 (concatenated numbers: 1.1.1, etc.), [0:1]

Hu Heading level for unnumbered heading (.HU) (4.3)
2 (.HU at the same level as .H 2), [0:7]

Hy Hyphenation control for body of document {3.4}
1 (automatic hyphenation on), [0:1]

L • Length of page {2.4}
66, [20:?] (11i, [2i:?] in *troff*)[23]

Li List indent {5.3.3.1}
5, [0:?]

Ls List spacing between items by level {5.3.3.1}
5, [0:5]

N • Numbering style {2.4}
0, [0:3]

O • Offset of page {2.4}
0, [0:?] (0.5i, [0i:?] in *troff*)[23]

P Page number, managed by PWB/MM {2.4}
0, [0:?]

Pi Paragraph indent {4.1}
5, [0:?]

Pt Paragraph type {4.1}
2 (paragraphs indented except after headings, lists, and displays), [0:2]

S • *Troff* default point size {2.4}
10, [6:36]

Si Standard indent for displays {7.1}
5, [0:?]

T • Type of *nroff* output device {2.4}
0, [0:2]

Tb Table counter {7.5}
0, [0:?], incremented by 1 for each .TB call.

U • Underlining style (*nroff*) for .H and .HU {2.4}
0 (continuous underline when possible), [0:1]

W • Width of page (line and title length) {2.4}
65, [10:1365] (6.5i, [2i:7.54i] in *troff*)[23]

23. For *nroff*, these values are *unscaled* numbers representing lines or character positions; for *troff*, these values must be *scaled*.